Capítulo 2

Rastreo o análisis léxico

La fase de rastreo, o **análisis léxico**, de un compilador tiene la tarea de leer el programa fuente como un archivo de caracteres y dividirlo en tokens. Los tokens son como las palabras de un lenguaje natural: cada token es una secuencia de caracteres que representa una unidad de información en el programa fuente. Ejemplos típicos de token son las **palabras reservadas**, como `if` y `while`, las cuales son cadenas fijas de letras; los **identificadores**, que son cadenas definidas por el usuario, compuestas por lo regular de letras y números, y que comienzan con una letra; los **símbolos especiales**, como los símbolos aritméticos + y *; además de algunos símbolos compuestos de múltiples caracteres, tales como > = y <>. En cada caso un token representa cierto patrón de caracteres que el analizador léxico reconoce, o ajusta desde el inicio de los caracteres de entrada restantes.

Como la tarea que realiza el analizador léxico es un caso especial de coincidencia de patrones, necesitamos estudiar métodos de especificación y reconocimiento de patrones en la medida en que se aplican al proceso de análisis léxico. Estos métodos son principalmente los de las **expresiones regulares** y los **autómatas finitos**. Sin embargo, un analizador léxico también es la parte del compilador que maneja la entrada del código fuente, y puesto que esta entrada a menudo involucra un importante gasto de tiempo, el analizador léxico debe funcionar de manera tan eficiente como sea posible. Por lo tanto, también necesitamos poner mucha atención a los detalles prácticos de la estructura del analizador léxico.

Dividiremos el estudio de las cuestiones del analizador léxico como sigue. En primer lugar, daremos una perspectiva general de la función de un analizador léxico y de las estructuras y conceptos involucrados. Enseguida, estudiaremos las expresiones regulares, una notación estándar para representar los patrones en cadenas que forman la estructura léxica de un lenguaje de programación. Continuaremos con el estudio de las máquinas de estados finitos, o autómatas finitos, las cuales representan algoritmos que permiten reconocer los patrones de cadenas dados por las expresiones regulares. También estudiaremos el proceso de construcción de los autómatas finitos fuera de las expresiones

regulares. Después volveremos a los métodos prácticos para escribir programas que implementen los procesos de reconocimiento representados por los autómatas finitos y estudiaremos una implementación completa de un analizador léxico para el lenguaje TINY. Por último, estudiaremos la manera en que se puede automatizar el proceso de producción de un programa analizador léxico por medio de un generador de analizador léxico, y repetiremos la implementación de un analizador léxico para TINY utilizando Lex, que es un generador de analizador léxico estándar disponible para su uso en Unix y otros sistemas.

2.1 EL PROCESO DEL ANÁLISIS LÉXICO

El trabajo del analizador léxico es leer los caracteres del código fuente y formarlos en unidades lógicas para que lo aborden las partes siguientes del compilador (generalmente el analizador sintáctico). Las unidades lógicas que genera el analizador léxico se denominan **tokens**, y formar caracteres en tokens es muy parecido a formar palabras a partir de caracteres en una oración en un lenguaje natural como el inglés o cualquier otro y decidir lo que cada palabra significa. En esto se asemeja a la tarea del deletreo.

Los tokens son entidades lógicas que por lo regular se definen como un tipo enumerado. Por ejemplo, pueden definirse en C como[1]

```
typedef enum
    {IF,THEN,ELSE,PLUS,MINUS,NUM,ID,...}
    TokenType;
```

Los tokens caen en diversas categorías, una de ellas la constituyen las **palabras reservadas**, como `IF` y `THEN`, que representan las cadenas de caracteres "if" y "then". Una segunda categoría es la de los **símbolos especiales**, como los símbolos aritméticos `MÁS` y `MENOS`, los que se representan con los caracteres "+" y "−". Finalmente, existen tokens que pueden representar cadenas de múltiples caracteres. Ejemplos de esto son `NUM` e `ID`, los cuales representan números e identificadores.

Los tokens como entidades lógicas se deben distinguir claramente de las cadenas de caracteres que representan. Por ejemplo, el token de la palabra reservada `IF` se debe distinguir de la cadena de caracteres "if" que representa. Para hacer clara la distinción, la cadena de caracteres representada por un token se denomina en ocasiones su **valor de cadena** o su **lexema**. Algunos tokens tienen sólo un lexema: las palabras reservadas tienen esta propiedad. No obstante, un token puede representar un número infinito de lexemas. Los identificadores, por ejemplo, están todos representados por el token simple `ID`, pero tienen muchos valores de cadena diferentes que representan sus nombres individuales. Estos nombres no se pueden pasar por alto, porque un compilador debe estar al tanto de ellos en una tabla de símbolos. Por consiguiente, un rastreador o analizador léxico también debe construir los valores de cadena de por lo menos algunos de los tokens. Cualquier valor asociado a un token

---

1. En un lenguaje sin tipos enumerados tendríamos que definir los tokens directamente como valores *numéricos simbólicos. Así, al estilo antiguo de C, en ocasiones se vería lo siguiente:*

```
#define IF 256
#define THEN 257
#define ELSE 258
...
```

(Estos números comienzan en 256 para evitar que se confundan con los valores ASCII numéricos.)

se denomina **atributo** del token, y el valor de cadena es un ejemplo de un atributo. Los tokens también pueden tener otros atributos. Por ejemplo, un token **NUM** puede tener un atributo de valor de cadena como "32767", que consta de cinco caracteres numéricos, pero también tendrá un atributo de valor numérico que consiste en el valor real de 32767 calculado a partir de su valor de cadena. En el caso de un token de símbolo especial como **MÁS** (**PLUS**), no sólo se tiene el valor de cadena "+", sino también la operación aritmética real + que está asociada con él mismo. En realidad, el símbolo del token mismo se puede ver simplemente como otro atributo, mientras que el token se puede visualizar como la colección de todos sus atributos.

Un analizador léxico necesita calcular al menos tantos atributos de un token como sean necesarios para permitir el procesamiento siguiente. Por ejemplo, se necesita calcular el valor de cadena de un token **NUM**, pero no es necesario calcular de inmediato su valor numérico, puesto que se puede calcular de su valor de cadena. Por otro lado, si se calcula su valor numérico, entonces se puede descartar su valor de cadena. En ocasiones el mismo analizador léxico puede realizar las operaciones necesarias para registrar un atributo en el lugar apropiado, o puede simplemente pasar el atributo a una fase posterior del compilador. Por ejemplo, un analizador léxico podría utilizar el valor de cadena de un identificador para introducirlo a la tabla de símbolos, o podría pasarlo para introducirlo en una etapa posterior.

Puesto que el analizador léxico posiblemente tendrá que calcular varios atributos para cada token, a menudo es útil recolectar todos los atributos en un solo tipo de datos estructurados, al que podríamos denominar como **registro de token**. Un registro así se podría declarar en C como

```
typedef struct
   { TokenType tokenval;
     char * stringval;
     int numval;
   } TokenRecord;
```

o posiblemente como una unión

```
typedef struct
   { TokenType tokenval;
     union
     { char * stringval;
       int numval;
     } attribute;
   } TokenRecord;
```

(lo que supone que el atributo de valor de cadena sólo es necesario para identificadores y el atributo de valor numérico sólo para números). Un arreglo más común es que el analizador léxico solamente devuelva el valor del token y coloque los otros atributos en variables donde se pueda tener acceso a ellos por otras partes del compilador.

Aunque la tarea del analizador léxico es convertir todo el programa fuente en una secuencia de tokens, pocas veces el analizador hará todo esto de una vez. En realidad, el analizador léxico funcionará bajo el control del analizador sintáctico, devolviendo el siguiente token simple desde la entrada bajo demanda mediante una función que tendrá una declaración similar a la declaración en el lenguaje C

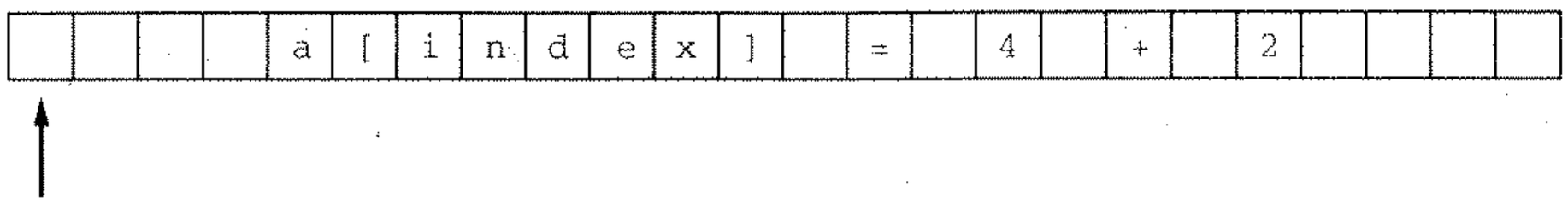
```
TokenType getToken(void);
```

La función **getToken** declarada de esta manera devolverá, cuando se le llame, el siguiente token desde la entrada, y además calculará atributos adicionales, como el valor de cadena del token. La cadena de caracteres de entrada por lo regular no tiene un parámetro para esta función, pero se conserva en un buffer (localidad de memoria intermedia) o se proporciona mediante las facilidades de entrada del sistema.

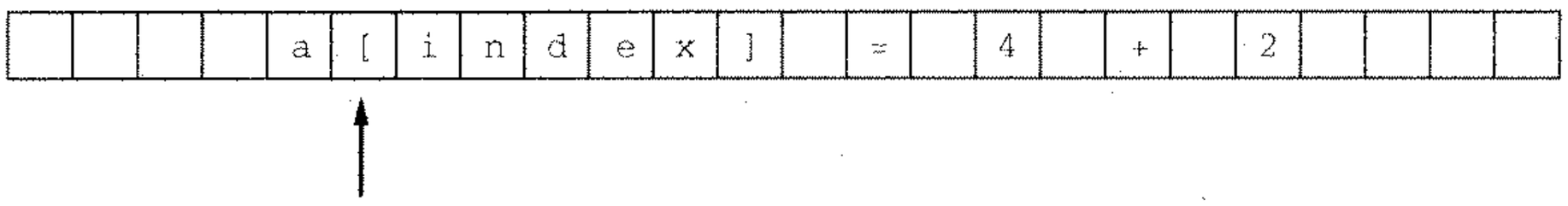
Como ejemplo del funcionamiento de **getToken** considere la siguiente línea de código fuente en C, que utilizamos como ejemplo en el capítulo 1:

```
a[index] = 4 + 2
```

Suponga que esta línea de código se almacena en un buffer de entrada como sigue, con el siguiente carácter de entrada indicado por la flecha:



Una llamada a **getToken** necesitará ahora saltarse los siguientes cuatro espacios en blanco, reconocer la cadena "a" compuesta del carácter único *a* como el token siguiente, y devolver el valor de token **ID** como el token siguiente, dejando el buffer de entrada como se aprecia a continuación:



De esta manera, una llamada posterior a **getToken** comenzará de nuevo el proceso de reconocimiento con el carácter de corchete izquierdo.

Volveremos ahora al estudio de métodos para definir y reconocer patrones en cadenas de caracteres.

2.2 EXPRESIONES REGULARES

Las expresiones regulares representan patrones de cadenas de caracteres. Una expresión regular *r* se encuentra completamente definida mediante el conjunto de cadenas con las que concuerda. Este conjunto se denomina **lenguaje generado por la expresión regular** y se escribe como $L(r)$. Aquí la palabra *lenguaje* se utiliza sólo para definir "conjunto de cadenas" y no tiene (por lo menos en esta etapa) una relación específica con un lenguaje de programación. Este lenguaje depende, en primer lugar, del conjunto de caracteres que se encuentra disponible. En general, estaremos hablando del conjunto de caracteres ASCII o de algún subconjunto del mismo. En ocasiones el conjunto será más general que el conjunto de caracteres ASCII, en cuyo caso los elementos del conjunto se describirán como **símbolos**. Este conjunto de símbolos legales se conoce como **alfabeto** y por lo general se representa mediante el símbolo griego Σ (sigma).

Una expresión regular *r* también contendrá caracteres del alfabeto, pero esos caracteres tendrán un significado diferente: en una expresión regular todos los símbolos indican *patrones*. En este capítulo distinguiremos el uso de un carácter como patrón escribiendo todo los patrones en negritas. De este modo, **a** es el carácter *a* usado como patrón.

Por último, una expresión regular *r* puede contener caracteres que tengan significados especiales. Este tipo de caracteres se llaman **metacaracteres** o **metasímbolos**, y por lo general no pueden ser caracteres legales en el alfabeto, porque no podríamos distinguir su uso como metacaracteres de su uso como miembros del alfabeto. Sin embargo, a menudo no es posible requerir tal exclusión, por lo que se debe utilizar una convención para diferenciar los dos usos posibles de un metacarácter. En muchas situaciones esto se realiza mediante el uso de un **carácter de escape** que "desactiva" el significado especial de un metacarácter. Unos caracteres de escape comunes son la diagonal inversa y las comillas. Advierta que los caracteres de escape, si también son caracteres legales en el alfabeto, son por sí mismos metacaracteres.

2.2.1 Definición de expresiones regulares

Ahora estamos en posición de describir el significado de las expresiones regulares al establecer cuáles lenguajes genera cada patrón. Haremos esto en varias etapas. Comenzaremos por describir el conjunto de expresiones regulares básicas, las cuales se componen de símbolos individuales. Continuaremos con la descripción de las operaciones que generan nuevas expresiones regulares a partir de las ya existentes. Esto es similar a la manera en que se construyen las expresiones aritméticas: las expresiones aritméticas básicas son los números, tales como 43 y 2.5. Entonces las operaciones aritméticas, como la suma y la multiplicación, se pueden utilizar para formar nuevas expresiones a partir de las existentes, como en el caso de 43 * 2.5 y 43 * 2.5 + 1.4.

El grupo de expresiones regulares que describiremos aquí es mínimo, ya que sólo contiene los metasímbolos y las operaciones esenciales. Después consideraremos extensiones a este conjunto mínimo.

Expresiones regulares básicas Éstas son precisamente los caracteres simples del alfabeto, los cuales se corresponden a sí mismos. Dado cualquier carácter a del alfabeto Σ, indicamos que la expresión regular **a** corresponde al carácter a escribiendo $L(\mathbf{a}) = \{a\}$. Existen otros dos símbolos que necesitaremos en situaciones especiales. Necesitamos poder indicar una concordancia con la **cadena vacía**, es decir, la cadena que no contiene ningún carácter. Utilizaremos el símbolo ε (épsilon) para denotar la cadena vacía, y definiremos el metasímbolo $\boldsymbol{\varepsilon}$ (ε en negritas) estableciendo que $L(\varepsilon) = \{\varepsilon\}$. También necesitaremos ocasionalmente ser capaces de describir un símbolo que corresponda a la ausencia de cadenas, es decir, cuyo lenguaje sea el **conjunto vacío**, el cual escribiremos como { }. Emplearemos para esto el símbolo $\boldsymbol{\phi}$ y escribiremos $L(\boldsymbol{\phi}) = \{\ \}$. Observe la diferencia entre { } y $\{\varepsilon\}$: el conjunto { } no contiene ninguna cadena, mientras que el conjunto $\{\varepsilon\}$ contiene la cadena simple que no se compone de ningún carácter.

Operaciones de expresiones regulares Existen tres operaciones básicas en las expresiones regulares: 1) selección entre alternativas, la cual se indica mediante el metacarácter | (barra vertical); 2) concatenación, que se indica mediante yuxtaposición (sin un metacarácter), y 3) repetición o "cerradura", la cual se indica mediante el metacarácter *. Analizaremos cada una por turno, proporcionando la construcción del conjunto correspondiente para los lenguajes de cadenas concordantes.

Selección entre alternativas Si r y s son expresiones regulares, entonces $r \mid s$ es una expresión regular que define cualquier cadena que concuerda con r o con s. En términos de lenguajes, el lenguaje de $r \mid s$ es la **unión** de los lenguajes de r y s, o $L(r \mid s) = L(r) \cup L(s)$. Como un ejemplo simple, considere la expresión regular **a|b**: ésta corresponde tanto al

carácter *a* como al carácter *b*, es decir, $L(\mathbf{a} \mid \mathbf{b}) = L(\mathbf{a}) \cup L(\mathbf{b}) = \{a\} \cup \{b\} = \{a, b\}$. Como segundo ejemplo, la expresión regular **a | ε** corresponde tanto al carácter simple *a* como a la cadena vacía (que no está compuesta por ningún carácter). En otras palabras, $L(\mathbf{a} \mid \boldsymbol{\varepsilon}) = \{a, \varepsilon\}$.

La selección se puede extender a más de una alternativa, de manera que, por ejemplo, $L(\mathbf{a} \mid \mathbf{b} \mid \mathbf{c} \mid \mathbf{d}) = \{a, b, c, d\}$. En ocasiones también escribiremos largas secuencias de selecciones con puntos, como en **a | b | . . . | z**, que corresponde a cualquiera de las letras minúsculas de la *a* a la *z*.

Concatenación La concatenación de dos expresiones regulares *r* y *s* se escribe como *rs*, y corresponde a cualquier cadena que sea la concatenación de dos cadenas, con la primera de ellas correspondiendo a *r* y la segunda correspondiendo a *s*. Por ejemplo, la expresión regular **ab** corresponde sólo a la cadena *ab*, mientras que la expresión regular **(a | b) c** corresponde a las cadenas *ac* y *bc*. (El uso de los paréntesis como metacaracteres en esta expresión regular se explicará en breve.)

Podemos describir el efecto de la concatenación en términos de lenguajes generados al definir la concatenación de dos conjuntos de cadenas. Dados dos conjuntos de cadenas S_1 y S_2, el conjunto concatenado de cadenas S_1S_2 es el conjunto de cadenas de S_1 complementado con todas las cadenas de S_2. Por ejemplo, si $S_1 = \{aa, b\}$ y $S_2 = \{a, bb\}$, entonces $S_1S_2 = \{aaa, aabb, ba, bbb\}$. Ahora la operación de concatenación para expresiones regulares se puede definir como sigue: $L(rs) = L(r)L(s)$. De esta manera (utilizando nuestro ejemplo anterior), $L(\mathbf{(a \mid b) c}) = L(\mathbf{a \mid b})L(\mathbf{c}) = \{a, b\}\{c\} = \{ac, bc\}$.

La concatenación también se puede extender a más de dos expresiones regulares: $L(r_1 r_2 \ldots r_n) = L(r_1)L(r_2) \ldots L(r_n)$ = el conjunto de cadenas formado al concatenar todas las cadenas de cada una de las $L(r_1), \ldots, L(r_n)$.

Repetición La operación de repetición de una expresión regular, denominada también en ocasiones **cerradura (de Kleene)**, se escribe *r**, donde *r* es una expresión regular. La expresión regular *r** corresponde a cualquier concatenación finita de cadenas, cada una de las cuales corresponde a *r*. Por ejemplo, **a*** corresponde a las cadenas ε, *a*, *aa*, *aaa*, ... (Concuerda con ε porque ε es la concatenación de *ninguna* cadena concordante con **a**.) Podemos definir la operación de repetición en términos de lenguajes generados definiendo, a su vez, una operación similar * para conjuntos de cadenas. Dado un conjunto *S* de cadenas, sea

$$S^* = \{\varepsilon\} \cup S \cup SS \cup SSS \cup \ldots$$

Ésta es una unión de conjuntos infinita, pero cada uno de sus elementos es una concatenación finita de cadenas de *S*. En ocasiones el conjunto S^* se escribe como sigue:

$$S^* = \bigcup_{n=0}^{\infty} S^n$$

donde $S^n = S. . .S$ es la concatenación de *S* por *n* veces. ($S^0 = \{\varepsilon\}$.)

Ahora podemos definir la operación de repetición para expresiones regulares como sigue:

$$L(r^*) = L(r)^*$$

Considere como ejemplo la expresión regular `(a|bb)*`. (De nueva cuenta, la razón de tener paréntesis como metacaracteres se explicará más adelante.) Esta expresión regular corresponde a cualquiera de las cadenas siguientes: ε, *a*, *bb*, *aa*, *abb*, *bba*, *bbbb*, *aaa*, *aabb* y así sucesivamente. En términos de lenguajes, $L($`(a|bb)*`$) = L($`a|bb`$)^* = \{a, bb\}^* = \{\varepsilon, a, bb, aa, abb, bba, bbbb, aaa, aabb, abba, abbbb, bbaa, \ldots\}$.

Precedencia de operaciones y el uso de los paréntesis La descripción precedente no toma en cuenta la cuestión de la precedencia de las operaciones de elección, concatenación y repetición. Por ejemplo, dada la expresión regular `a|b*`, ¿deberíamos interpretar esto como `(a|b)*` o como `a|(b*)`? (Existe una diferencia importante, puesto que $L($`(a|b)*`$) = \{\varepsilon, a, b, aa, ab, ba, bb, \ldots\}$, mientras que $L($`a|(b*)`$) = \{\varepsilon, a, b, bb, bbb, \ldots\}$.) La convención estándar es que la repetición debería tener mayor precedencia, por lo tanto, la segunda interpretación es la correcta. En realidad, entre las tres operaciones, se le da al `*` la precedencia más alta, a la concatenación se le da la precedencia que sigue y a la | se le otorga la precedencia más baja. De este modo, por ejemplo, `a|bc*` se interpreta como `a|(b(c*))`, mientras que `ab|c*d` se interpreta como `(ab)|((c*)d)`.

Cuando deseemos indicar una precedencia diferente, debemos usar paréntesis para hacerlo. Ésta es la razón por la que tuvimos que escribir `(a|b)c` para indicar que la operación de elección debería tener mayor precedencia que la concatenación, ya que de otro modo `a|bc` se interpretaría como si correspondiera tanto a *a* como a *bc*. De manera similar, `(a|bb)*` se interpretaría sin los paréntesis como `a|bb*`, lo que corresponde a *a*, *b*, *bb*, *bbb*, ... Los paréntesis aquí se usan igual que en aritmética, donde (3 + 4) * 5 = 35, pero 3 + 4 * 5 = 23, ya que se supone que * tiene precedencia más alta que +.

Nombres para expresiones regulares A menudo es útil como una forma de simplificar la notación proporcionar un nombre para una expresión regular larga, de modo que no tengamos que escribir la expresión misma cada vez que deseemos utilizarla. Por ejemplo, si deseáramos desarrollar una expresión regular para una secuencia de uno o más dígitos numéricos, entonces escribiríamos

```
(0|1|2|...|9)(0|1|2|...|9)*
```

o podríamos escribir

```
dígito dígito*
```

donde

```
dígito = 0|1|2|...|9
```

es una **definición regular** del nombre *dígito*.

El uso de una definición regular es muy conveniente, pero introduce la complicación agregada de que el nombre mismo se convierta en un metasímbolo y se deba encontrar un significado para distinguirlo de la concatenación de sus caracteres. En nuestro caso hicimos esa distinción al utilizar letra cursiva para el nombre. Advierta que no se debe emplear el nombre del término en su propia definición (es decir, de manera recursiva): debemos poder eliminar nombres reemplazándolos sucesivamente con las expresiones regulares para las que se establecieron.

Antes de considerar una serie de ejemplos para elaborar nuestra definición de expresiones regulares, reuniremos todas las piezas de la definición de una expresión regular.

Definición

Una **expresión regular** es una de las siguientes:

1. Una expresión regular **básica** constituida por un solo carácter **a**, donde *a* proviene de un alfabeto Σ de caracteres legales; el metacarácter **ε**; o el metacarácter **ϕ**. En el primer caso, $L(\mathbf{a}) = \{a\}$; en el segundo, $L(\boldsymbol{\varepsilon}) = \{\varepsilon\}$; en el tercero, $L(\boldsymbol{\phi}) = \{\}$.
2. Una expresión de la forma ***r*|*s***, donde *r* y *s* son expresiones regulares. En este caso, $L(\boldsymbol{r}|\boldsymbol{s}) = L(r) \cup L(s)$.
3. Una expresión de la forma *rs*, donde *r* y *s* son expresiones regulares. En este caso, $L(rs) = L(r)L(s)$.
4. Una expresión de la forma *r**, donde *r* es una expresión regular. En este caso, $L(r^*) = L(r)^*$.
5. Una expresión de la forma **(***r***)**, donde *r* es una expresión regular. En este caso, $L((r)) = L(r)$. De este modo, los paréntesis no cambian el lenguaje, sólo se utilizan para ajustar la precedencia de las operaciones.

Advertimos que, en esta definición, la precedencia de las operaciones en (2), (3) y (4) está en el orden inverso de su enumeración; es decir, | tiene precedencia más baja que la concatenación, y ésta tiene una precedencia más baja que el asterisco *. También advertimos que esta definición proporciona un significado de metacarácter a los seis símbolos **ϕ**, **ε**, |, *, (,).

En lo que resta de esta sección consideraremos una serie de ejemplos diseñados para explicar con más detalles la definición que acabamos de dar. Éstos son algo artificiales, ya que por lo general no aparecen como descripciones de token en un lenguaje de programación. En la sección 2.2.3 consideraremos algunas expresiones regulares comunes que aparecen con frecuencia como tokens en lenguajes de programación.

En los ejemplos siguientes por lo regular se incluye una descripción en idioma coloquial de las cadenas que se harán corresponder, y la tarea es traducir la descripción a una expresión regular. Esta situación, en la que un manual de lenguaje contiene descripciones de los tokens, es la que con más frecuencia enfrentan quienes escriben compiladores. En ocasiones puede ser necesario invertir la dirección, es decir, desplazarse de una expresión regular hacia una descripción en lenguaje coloquial, de modo que también incluiremos algunos ejercicios de esta clase.

Ejemplo 2.1

Consideremos el alfabeto simple constituido por sólo tres caracteres alfabéticos: $\Sigma = \{a, b, c\}$. También el conjunto de todas las cadenas en este alfabeto que contengan exactamente una *b*. Este conjunto es generado por la expresión regular

```
(a|c)*b(a|c)*
```

Advierta que, aunque **b** aparece en el centro de la expresión regular, la letra *b* no necesita estar en el centro de la cadena que se desea definir. En realidad, la repetición de *a* o *c* antes y después de la *b* puede presentarse en diferentes números de veces. Por consiguiente, todas las cadenas siguientes están generadas mediante la expresión regular anterior: *b, abc, abaca, baaaac, ccbaca, ccccccb.* §

Ejemplo 2.2

Con el mismo alfabeto que antes, considere el conjunto de todas las cadenas que contienen como máximo una *b*. Una expresión regular para este conjunto se puede obtener utilizando la solución al ejemplo anterior como una alternativa (definiendo aquellas cadenas con exactamente una *b*) y la expresión regular `(a|c)*` como la otra alternativa (definiendo los casos sin *b* en todo). De este modo, tenemos la solución siguiente:

```
(a|c)*|(a|c)*b(a|c)*
```

Una solución alternativa sería permitir que *b* o la cadena vacía apareciera entre las dos repeticiones de *a* o *c*:

```
(a|c)*(b|ε)(a|c)*
```

Este ejemplo plantea un punto importante acerca de las expresiones regulares: el mismo lenguaje se puede generar mediante muchas expresiones regulares diferentes. Por lo general, intentamos encontrar una expresión regular tan simple como sea posible para describir un conjunto de cadenas, aunque nunca intentaremos demostrar que encontramos, de hecho, la "más simple": la más breve por ejemplo. Existen dos razones para esto. La primera es que raramente se presenta en situaciones prácticas, donde por lo regular existe una solución estándar "más simple". La segunda es que cuando estudiemos métodos para reconocer expresiones regulares, los algoritmos tendrán que poder simplificar el proceso de reconocimiento sin molestarse en simplificar primero la expresión regular. §

Ejemplo 2.3

Consideremos el conjunto de cadenas *S* sobre el alfabeto $\Sigma = \{a, b\}$ compuesto de una *b* simple rodeada por el mismo número de *a*:

$$S = \{b, aba, aabaa, aaabaaa, \ldots\} = \{a^n b a^n | n \neq 0\}$$

Este conjunto no se puede describir mediante una expresión regular. La razón es que la única operación de repetición que tenemos es la operación de cerradura `*`, la cual permite cualquier número de repeticiones. De modo que si escribimos la expresión `a*ba*` (lo más cercano que podemos obtener en el caso de una expresión regular para *S*), no hay garantía de que el número de *a* antes y después de la *b* será el mismo. Expresamos esto al decir que "las expresiones regulares no pueden contar". Sin embargo, para proporcionar una demostración matemática de este hecho, requeriríamos utilizar un famoso teorema acerca de las expresiones regulares conocido como **lema de la extracción** (**pumping lemma**), que se estudia en la teoría de autómatas, pero que aquí ya no volveremos a mencionar.

Es evidente que no todos los conjuntos de cadenas que podemos describir en términos simples se pueden generar mediante expresiones regulares. Por consiguiente, un conjunto de cadenas que *es* el lenguaje para una expresión regular se distingue de otros conjuntos al denominarlo **conjunto regular**. De cuando en cuando aparecen conjuntos no regulares como cadenas en lenguajes de programación que necesitan ser reconocidos por un analizador léxico, los cuales por lo regular son abortados cuando surgen. Retomaremos este tema de manera breve en la sección en que se abordan las consideraciones para el analizador léxico práctico. §

Ejemplo 2.4

Consideremos las cadenas en el alfabeto $\Sigma = \{a, b, c\}$ que no contienen dos *b* consecutivas. De modo que, entre cualesquiera dos *b*, debe haber por lo menos una *a* o una *c*. Construiremos

una expresión regular para este conjunto en varias etapas. En primer lugar, podemos obligar a que una *a* o *c* se presenten *después* de cualquier *b* al escribir

```
(b(a|c))*
```

Podemos combinar esto con la expresión `(a|c)*`, la cual define cadenas que no tienen *b* alguna, y escribimos

```
((a|c)*|(b(a|c))*)*
```

o, advirtiendo que (*r** | *s**)* corresponde con las mismas cadenas que (*r* | *s*)*

```
((a|c)|(b(a|c)))*
```

o

```
(a|c|ba|bc)*
```

(¡Cuidado! Ésta todavía no es la respuesta correcta.)

El lenguaje generado por esta expresión regular tiene, en realidad, la propiedad que buscamos, a saber, que no haya dos *b* consecutivas (pero aún no es lo bastante correcta). En ocasiones podremos demostrar tales aseveraciones, así que esbozaremos una demostración de que todas las cadenas en *L*(`(a|c|ba|bc)*`) no contienen dos *b* consecutivas. La demostración es por inducción sobre la longitud de la cadena (es decir, el número de caracteres en la cadena). Es evidente que esto es verdadero para todas las cadenas de longitud 0, 1 o 2: estas cadenas son precisamente las cadenas ε, *a*, *c*, *aa*, *ac*, *ca*, *cc*, *ba*, *bc*. Ahora supongamos que es verdadero para todas las cadenas en el lenguaje con una longitud de $i < n$, y sea *s* una cadena en el lenguaje con longitud $n > 2$. Entonces, *s* contiene más de una de las cadenas no-ε anteriormente enumeradas, de modo que $s = s_1 s_2$, donde s_1 y s_2 también se encuentran en el lenguaje y no son ε. Por lo tanto, mediante la hipótesis de inducción, tanto s_1 como s_2 no tienen dos *b* consecutivas. Por consiguiente, la única manera de que *s* misma pudiera tener dos *b* consecutivas sería que s_1 finalizara con una *b* y que s_2 comenzara con una *b*. Pero esto es imposible, porque ninguna cadena en el lenguaje puede finalizar con una *b*.

Este último hecho que utilizamos en el esbozo de la demostración (o sea, que ninguna cadena generada mediante la expresión regular anterior puede finalizar con una *b*) también muestra por qué nuestra solución todavía no es bastante correcta: no genera las cadenas *b*, *ab* y *cb*, que no contienen dos *b* consecutivas. Arreglaremos esto agregando una *b* opcional rezagada de la manera siguiente:

```
(a|c|ba|bc)*(b|ε)
```

Advierta que la imagen especular de esta expresión regular también genera el lenguaje dado

```
(b|ε)(a|c|ab|cb)*
```

También podríamos generar este mismo lenguaje escribiendo

```
(notb|b notb)*(b|ε)
```

donde ***notb*** `= a|c`. Éste es un ejemplo del uso de un nombre para una subexpresión. De hecho, esta solución es preferible en los casos en que el alfabeto es grande, puesto que la definición de ***notb*** se puede ajustar para incluir todos los caracteres, excepto *b*, sin complicar la expresión original. §

Ejemplo 2.5

En este ejemplo proporcionamos la expresión regular y solicitamos determinar una descripción concisa en español del lenguaje que genera. Considere el alfabeto $\Sigma = \{a, b, c\}$ y la expresión regular

```
((b|c)*a(b|c)*a)*(b|c)*
```

Esto genera el lenguaje de todas las cadenas que contengan un número par de *a*. Para ver esto considere la expresión dentro de la repetición exterior izquierda:

```
(b|c)*a(b|c)*a
```

Esto genera aquellas cadenas que finalizan en *a* y que contienen exactamente dos *a* (cualquier número de *b* y de *c* puede aparecer antes o entre las dos *a*). La repetición de estas cadenas nos da todas las cadenas que finalizan en *a* cuyo número de *a* es un múltiplo de 2 (es decir, par). Al añadir la repetición `(b|c)*` al final (como en el ejemplo anterior) obtenemos el resultado deseado.

Advertimos que esta expresión regular también se podría escribir como

```
(nota* a nota* a)* nota*
```

§

2.2.2 Extensiones para las expresiones regulares

Formulamos una definición para las expresiones regulares que emplean un conjunto mínimo de operaciones comunes a todas las aplicaciones, y podríamos limitarnos a utilizar sólo las tres operaciones básicas (junto con paréntesis) en todos nuestros ejemplos. Sin embargo, ya vimos en los ejemplos anteriores que escribir expresiones regulares utilizando sólo estos operadores en ocasiones es poco manejable, ya que se crean expresiones regulares que son más complicadas de lo que serían si se dispusiera de un conjunto de operaciones más expresivo. Por ejemplo, sería útil tener una notación para una coincidencia de cualquier carácter (ahora tenemos que enumerar todos los caracteres en el alfabeto como una larga alternativa). Además, ayudaría tener una expresión regular para una gama de caracteres y una expresión regular para todos los caracteres excepto uno.

En los párrafos siguientes describiremos algunas extensiones a las expresiones regulares estándar ya analizadas, con los correspondientes metasímbolos nuevos, que cubren éstas y situaciones comunes semejantes. En la mayoría de estos casos no existe una terminología común, de modo que utilizaremos una notación similar a la empleada por el generador de analizadores léxicos Lex, el cual se describe más adelante en este capítulo. En realidad, muchas de las situaciones que describiremos aparecerán de nuevo en nuestra descripción de Lex. No obstante, no todas las aplicaciones que utilizan expresiones regulares incluirán estas operaciones, e incluso aunque lo hicieran, se puede emplear una notación diferente.

Ahora pasaremos a nuestra lista de nuevas operaciones.

UNA O MÁS REPETICIONES

Dada una expresión regular *r*, la repetición de *r* se describe utilizando la operación de cerradura estándar, que se escribe *r**. Esto permite que *r* se repita 0 o más veces. Una situación típica que surge es la necesidad de *una* o más repeticiones en lugar de ninguna, lo que garantiza que aparece por lo menos una cadena correspondiente a *r*, y no permite la cadena vacía ε. Un ejemplo es el de un número natural, donde queremos una secuencia de dígitos, pero deseamos que por lo menos aparezca uno. Por ejemplo, si deseamos definir números binarios, podríamos escribir `(0|1)*`, pero esto también

coincidirá con la cadena vacía, la cual no es un número. Por supuesto, podríamos escribir

```
(0|1)(0|1)*
```

pero esta situación se presenta con tanta frecuencia que se desarrolló para ella una notación relativamente estándar en la que se utiliza + en lugar de *: *r*+, que indica una o más repeticiones de *r*. De este modo, nuestra expresión regular anterior para números binarios puede escribirse ahora como

```
(0|1)+
```

CUALQUIER CARÁCTER

Una situación común es la necesidad de generar cualquier carácter en el alfabeto. Sin una operación especial esto requiere que todo carácter en el alfabeto sea enumerado en una alternativa. Un metacarácter típico que se utiliza para expresar una concordancia de cualquier carácter es el punto "`.`", el cual no requiere que el alfabeto se escriba realmente en forma extendida. Con este metacarácter podemos escribir una expresión regular para todas las cadenas que contengan al menos una *b* como se muestra a continuación:

```
.*b.*
```

UN INTERVALO DE CARACTERES

A menudo necesitamos escribir un intervalo de caracteres, como el de todas las letras minúsculas o el de todos los dígitos. Hasta ahora hemos hecho esto utilizando la notación `a|b|...|z` para las letras minúsculas o `0|1|...|9` para los dígitos. Una alternativa es tener una notación especial para esta situación, y una que es común es la de emplear corchetes y un guión, como en `[a-z]` para las letras minúsculas y `[0-9]` para los dígitos. Esto también se puede emplear para alternativas individuales, de modo que `a|b|c` puede escribirse como `[abc]`. También se pueden incluir los intervalos múltiples, de manera que `[a-zA-Z]` representa todas las letras minúsculas y mayúsculas. Esta notación general se conoce como **clases de caracteres**. Advierta que esta notación puede depender del orden subyacente del conjunto de caracteres. Por ejemplo, al escribir `[A-Z]` se supone que los caracteres *B*, *C*, y los demás vienen entre los caracteres *A* y *Z* (una suposición razonable) y que sólo los caracteres en mayúsculas están entre *A* y *Z* (algo que es verdadero para el conjunto de caracteres ASCII). Sin embargo, al escribir `[A-z]` *no* se definirán los mismos caracteres que para `[A-Za-z]`, incluso en el caso del conjunto de caracteres ASCII.

CUALQUIER CARÁCTER QUE NO ESTÉ EN UN CONJUNTO DADO

Como hemos visto, a menudo es de utilidad poder excluir un carácter simple del conjunto de caracteres por generar. Esto se puede conseguir al diseñar un metacarácter para indicar la operación de negación ("not") o complementaria sobre un conjunto de alternativas. Por ejemplo, un carácter estándar que representa la negación en lógica es la "tilde" ~, y podríamos escribir una expresión regular para un carácter en el alfabeto que no sea *a* como `~a` y un carácter que no sea *a*, ni *b*, ni *c*, como

```
~(a|b|c)
```

Una alternativa para esta notación se emplea en Lex, donde el carácter "carat" ^ se utiliza en conjunto con las clases de caracteres que acabamos de describir para la formación de

complementos. Por ejemplo, cualquier carácter que no sea *a* se escribe como `[^a]`, mientras que cualquier carácter que no sea *a*, ni *b* ni *c* se escribe como

```
[^abc]
```

SUBEXPRESIONES OPCIONALES

Por último, un suceso que se presenta comúnmente es el de cadenas que contienen partes opcionales que pueden o no aparecer en cualquier cadena en particular. Por ejemplo, un número puede o no tener un signo inicial, tal como + o -. Podemos emplear alternativas para expresar esto como en las definiciones regulares

```
natural = [0-9]+
naturalconSigno = natural | + natural | - natural
```

Esto se puede convertir rápidamente en algo voluminoso, e introduciremos el metacarácter de signo de interrogación *r*`?` para indicar que las cadenas que coincidan con *r* son opcionales (o que están presentes 0 o 1 copias de *r*). De este modo, el ejemplo del signo inicial se convierte en

```
natural = [0-9]+
naturalconSigno = (+|-)? natural
```

2.2.3 Expresiones regulares para tokens de lenguajes de programación

Los tokens de lenguajes de programación tienden a caer dentro de varias categorías limitadas que son bastante estandarizadas a través de muchos lenguajes de programación diferentes. Una categoría es la de las **palabras reservadas**, en ocasiones también conocidas como **palabras clave**, que son cadenas fijas de caracteres alfabéticos que tienen un significado especial en el lenguaje. Los ejemplos incluyen `if`, `while` y `do` en lenguajes como Pascal, C y Ada. Otra categoría se compone de los **símbolos especiales**, que incluyen operadores aritméticos, de asignación y de igualdad. Éstos pueden ser un carácter simple, tal como =, o múltiples caracteres o compuestos, tales como `:=` o `++`. Una tercera categoría se compone de los **identificadores**, que por lo común se definen como secuencias de letras y dígitos que comienzan con una letra. Una categoría final se compone de **literales** o **constantes**, que incluyen constantes numéricas tales como 42 y 3.14159, literales de cadena como "hola, mundo" y caracteres tales como "a" y "b". Aquí describiremos algunas de estas expresiones regulares típicas y analizaremos algunas otras cuestiones relacionadas con el reconocimiento de tokens. Más detalles acerca de las cuestiones de reconocimiento práctico aparecen posteriormente en el capítulo.

Números Los números pueden ser sólo secuencias de dígitos (números naturales), o números decimales, o números con un exponente (indicado mediante una "e" o "E"). Por ejemplo, 2.71E-2 representa el número .0271. Podemos escribir definiciones regulares para esos números como se ve a continuación:

```
nat = [0-9]+
natconSigno = (+|-)? nat
número = natconSigno("." nat)?(E natconSigno)?
```

Aquí escribimos el punto decimal entre comillas para enfatizar que debería ser generado directamente y no interpretado como un metacarácter.

Identificadores y palabras reservadas Las palabras reservadas son las más simples de escribir como expresiones regulares: están representadas por sus secuencias fijas de caracteres. Si quisiéramos recolectar todas las palabras reservadas en una definición, escribiríamos algo como

```
reservada = if | while | do | ...
```

Los identificadores, por otra parte, son cadenas de caracteres que no son fijas. Un identificador debe comenzar, por lo común, con una letra y contener sólo letras y dígitos. Podemos expresar esto en términos de definiciones regulares como sigue:

```
letra = [a-zA-Z]
dígito = [0-9]
identificador = letra(letra|dígito)*
```

Comentarios Los comentarios por lo regular se ignoran durante el proceso del análisis léxico.[2] No obstante, un analizador léxico debe reconocer los comentarios y descartarlos. Por consiguiente, necesitaremos escribir expresiones regulares para comentarios, aun cuando un analizador léxico pueda no tener un token constante explícito (podríamos llamar a estos **pseudotokens**). Los comentarios pueden tener varias formas diferentes. Suelen ser de formato libre o estar rodeados de delimitadores tales como

```
{éste es un comentario de Pascal}
/* éste es un comentario de C */
```

o comenzar con un carácter o caracteres especificados y continuar hasta el final de la línea, como en

```
; éste es un comentario de Scheme
-- éste es un comentario de Ada
```

No es difícil escribir una expresión regular para comentarios que tenga delimitadores de carácter simple, tal como el comentario de Pascal, o para aquellos que partan de algún(os) carácter(es) especificado(s) hasta el final de la línea. Por ejemplo, el caso del comentario de Pascal puede escribirse como

```
{(~})*}
```

donde escribimos `~}` para indicar "not }" y donde partimos del supuesto de que el carácter `}` no tiene significado como un metacarácter. (Una expresión diferente debe escribirse para Lex, la cual analizaremos más adelante en este capítulo.) De manera similar, un comentario en Ada se puede hacer coincidir mediante la expresión regular

```
--(~nuevalínea)*
```

---

2. En ocasiones pueden contener directivas de compilador.

en la cual suponemos que **`nuevalínea`** corresponde al final de una línea (lo que se escribe como `\n` en muchos sistemas), que el carácter "`-`" no tiene significado como un metacarácter, y que el final de la línea no está incluido en el comentario mismo. (Veremos cómo escribir esto en Lex en la sección 2.6.)

Es mucho más difícil escribir una expresión regular para el caso de los delimitadores que tienen más de un carácter de longitud, tal como los comentarios de C. Para ver esto considere el conjunto de cadenas *ba. . .* (ausencia de apariciones de *ab*). . .*ab* (utilizamos *ba. . .ab* en lugar de los delimitadores de C /*. . .*/, ya que el asterisco, y en ocasiones la diagonal, es un metacarácter que requiere de un manejo especial). No podemos escribir simplemente

```
ba(~(ab))*ab
```

porque el operador "not" por lo regular está restringido a caracteres simples en lugar de cadenas de caracteres. Podemos intentar escribir una definición para `~(ab)` utilizando `~a`, `~b` y `~(a|b)`, pero esto es no trivial. Una solución es

```
b*(a*~(a|b)b*)*a*
```

pero es difícil de leer (y de demostrar correctamente). De este modo, una expresión regular para los comentarios de C es tan complicada que casi nunca se escribe en la práctica. De hecho, este caso se suele manejar mediante métodos ex profeso en los analizadores léxicos reales, lo que veremos más adelante en este capítulo.

Por último, otra complicación en el reconocimiento de los comentarios es que, en algunos lenguajes de programación, los comentarios pueden estar anidados. Por ejemplo, Modula-2 permite comentarios de la forma

```
(* esto es (* un comentario de *) en Modula-2 *)
```

Los delimitadores de comentario deben estar exactamente pareados en dichos comentarios anidados, de manera que lo que sigue no es un comentario legal de Modula-2:

```
(* esto es (* ilegal en Modula-2 *))
```

La anidación de los comentarios requiere que el analizador léxico cuente el número de los delimitadores. Pero advertimos en el ejemplo 2.3 (sección 2.2.1) que las expresiones regulares no pueden expresar operaciones de conteo. En la práctica utilizamos un esquema de contador simple como una solución adecuada para este problema (véanse los ejercicios).

Ambigüedad, espacios en blanco y búsqueda hacia delante A menudo en la descripción de los tokens de lenguajes de programación utilizando expresiones regulares, algunas cadenas se pueden definir mediante varias expresiones regulares diferentes. Por ejemplo, cadenas tales como `if` y `while` podrían ser identificadores o palabras clave. De manera semejante, la cadena `<>` se podría interpretar como la representación de dos tokens ("menor que" y "mayor que") o como un token simple ("no es igual a"). Una definición de lenguaje de programación debe establecer cuál interpretación se observará, y las expresiones regulares por sí mismas no pueden hacer esto. En realidad, una definición de lenguaje debe proporcionar **reglas de no ambigüedad** que implicarán cuál significado es el conveniente para cada uno de tales casos.

Dos reglas típicas que manejan los ejemplos que se acaban de dar son las siguientes. La primera establece que, cuando una cadena puede ser un identificador o una palabra clave, se prefiere por lo general la interpretación como palabra clave. Esto se da a entender mediante el uso del término **palabra reservada**, lo que quiere decir que es simplemente una palabra clave que no puede ser también un identificador. La segunda establece que, cuando una

cadena puede ser un token simple o una secuencia de varios tokens, por lo común se prefiere la interpretación del token simple. Esta preferencia se conoce a menudo como el **principio de la subcadena más larga**: la cadena más larga de caracteres que podrían constituir un token simple en cualquier punto se supone que representa el siguiente token.[3]

Una cuestión que surge con el uso del principio de la subcadena más larga es la cuestión de los **delimitadores de token**, o caracteres que implican que una cadena más larga en el punto donde aparecen no puede representar un token. Los caracteres que son parte no ambigua de otros tokens son delimitadores. Por ejemplo, en la cadena `xtemp=ytemp`, el signo de igualdad delimita el identificador `xtemp`, porque = no puede aparecer como parte de un identificador. Los espacios en blanco, los retornos de línea y los caracteres de tabulación generalmente también se asumen como delimitadores de token: `while x ...` se interpreta entonces como compuesto de dos tokens que representan la palabra reservada `while` y el identificador de nombre `x`, puesto que un espacio en blanco separa las dos cadenas de caracteres. En esta situación a menudo es útil definir un pseudotoken de espacio en blanco, similar al pseudotoken de comentario, que sólo sirve al analizador léxico de manera interna para distinguir otros tokens. En realidad, los comentarios mismos por lo regular sirven como delimitadores, de manera que, por ejemplo, el fragmento de código en lenguaje C

```
do/**/if
```

representa las dos palabras reservadas `do` e `if` más que el identificador `doif`.

Una definición típica del pseudotoken de espacio en blanco en un lenguaje de programación es

```
espacioenblanco = (nuevalínea|blanco|tabulación|comentario)+
```

donde los identificadores a la derecha representan los caracteres o cadenas apropiados. Advierta que, además de actuar como un delimitador de token, el espacio en blanco por lo regular no se toma en cuenta. Los lenguajes que especifican este comportamiento se denominan de **formato libre**. Las alternativas al formato libre incluyen el formato fijo de unos cuantos lenguajes como FORTRAN y diversos usos de la sangría en el texto, tal como la **regla de fuera de lugar** (véase la sección de notas y referencias). Un analizador léxico para un lenguaje de formato libre debe descartar el espacio en blanco después de verificar cualquier efecto de delimitación del token.

Los delimitadores terminan las cadenas que forman el token pero no son parte del token mismo. De este modo, un analizador léxico se debe ocupar del problema de la **búsqueda hacia delante**: cuando encuentra un delimitador debe arreglar que éste no se elimine del resto de la entrada, ya sea devolviéndolo a la cadena de entrada ("respaldándolo") o mirando hacia delante antes de eliminar el carácter de la entrada. En la mayoría de los casos sólo se necesita hacer esto para un carácter simple ("búsqueda hacia delante de carácter simple"). Por ejemplo, en la cadena `xtemp=ytemp`, el final del identificador `xtemp` se encuentra cuando se halla el =, y el signo = debe permanecer en la entrada, ya que representa el siguiente token a reconocer. Advierta también que es posible que no se necesite la búsqueda hacia delante para reconocer un token. Por ejemplo, el signo de igualdad puede ser el único token que comienza con el carácter =, en cuyo caso se puede reconocer de inmediato sin consultar el carácter siguiente.

En ocasiones un lenguaje puede requerir más que la búsqueda hacia delante de carácter simple, y el analizador léxico debe estar preparado para respaldar posiblemente de manera arbitraria muchos caracteres. En ese caso, el almacenamiento en memoria intermedia de los caracteres de entrada y la marca de lugares para un retroseguimiento se convierten en el diseño de un analizador léxico. (Algunas de estas preguntas se abordan más adelante en este capítulo.)

---

3. En ocasiones esto se denomina principio del "máximo bocado".

FORTRAN es un buen ejemplo de un lenguaje que viola muchos de los principios que acabamos de analizar. Éste es un lenguaje de formato fijo en el cual el espacio en blanco se elimina por medio de un preprocesador antes que comience la traducción. Por consiguiente, la línea en FORTRAN

```
I F ( X 2 . EQ. 0) THE N
```

aparecería para un compilador como

```
IF(X2.EQ.0)THEN
```

de manera que el espacio en blanco ya no funciona como un delimitador. Tampoco hay palabras reservadas en FORTRAN, de modo que todas las palabras clave también pueden ser identificadores, y la posición de la cadena de caracteres en cada línea de entrada es importante para determinar el token que será reconocido. Por ejemplo, la siguiente línea de código es perfectamente correcta en FORTRAN:

```
IF(IF.EQ.0)THENTHEN=1.0
```

Los primeros **IF** y **THEN** son palabras clave, mientras que los segundos **IF** y **THEN** son identificadores que representan variables. El efecto de esto es que un analizador léxico de FORTRAN debe poder retroceder a posiciones arbitrarias dentro de una línea de código. Consideremos, en concreto, el siguiente bien conocido ejemplo:

```
DO99I=1,10
```

Esto inicia un ciclo que comprende el código subsiguiente hasta la línea cuyo número es 99, con el mismo efecto que el Pascal `for i := 1 to 10`. Por otra parte, al cambiar la coma por un punto

```
DO99I=1.10
```

cambia el significado del código por completo: esto asigna el valor 1.1 a la variable con nombre **DO99I**. De este modo, un analizador léxico no puede concluir que el **DO** inicial es una palabra clave hasta que alcance el signo de coma (o el punto), en cuyo caso puede ser obligado a retroceder hacia el principio de la línea y comenzar de nuevo.

2.3 AUTÓMATAS FINITOS

Los autómatas finitos, o máquinas de estados finitos, son una manera matemática para describir clases particulares de algoritmos (o "máquinas"). En particular, los autómatas finitos se pueden utilizar para describir el proceso de reconocimiento de patrones en cadenas de entrada, y de este modo se pueden utilizar para construir analizadores léxicos. Por supuesto, también existe una fuerte relación entre los autómatas finitos y las expresiones regulares, y veremos en la sección siguiente cómo construir un autómata finito a partir de una expresión regular. Sin embargo, antes de comenzar nuestro estudio de los autómatas finitos de manera apropiada, consideraremos un ejemplo explicativo.

El patrón para identificadores como se define comúnmente en los lenguajes de programación está dado por la siguiente definición regular (supondremos que **letra** y **dígito** ya se definieron):

```
identificador = letra(letra|dígito)*
```

Esto representa una cadena que comienza con una letra y continúa con cualquier secuencia de letras y/o dígitos. El proceso de reconocer una cadena así se puede describir mediante el diagrama de la figura 2.1.

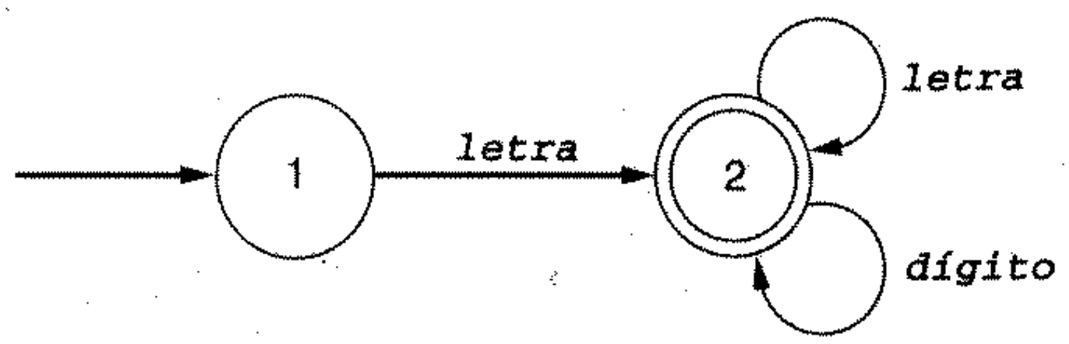


Figura 2.1
Un autómata finito para identificadores

En este diagrama los círculos numerados 1 y 2 representan **estados**, que son localidades en el proceso de reconocimiento que registran cuánto del patrón ya se ha visto. Las líneas con flechas representan **transiciones** que registran un cambio de un estado a otro en una coincidencia del carácter o caracteres mediante los cuales son etiquetados. En el diagrama de muestra, el estado 1 es el **estado de inicio**, o el estado en el que comienza el proceso de reconocimiento. Por convención, el estado de inicio se indica dibujando una línea con flechas sin etiqueta que proviene de "de ninguna parte". El estado 2 representa el punto en el cual se ha igualado una sola letra (lo que se indica mediante la transición del estado 1 al estado 2 etiquetada con `letra`). Una vez en el estado 2, cualquier número de letras y/o dígitos se puede ver, y una coincidencia de éstos nos regresa al estado 2. Los estados que representan el fin del proceso de reconocimiento, en los cuales podemos declarar un éxito, se denominan **estados de aceptación**, y se indican dibujando un borde con línea doble alrededor del estado en el diagrama. Puede haber más de uno de éstos. En el diagrama de muestra el estado 2 es un estado de aceptación, lo cual indica que, después que cede una letra, cualquier secuencia de letras y dígitos subsiguiente (incluyendo la ausencia de todas) representa un identificador legal.

El proceso de reconocimiento de una cadena de caracteres real como un identificador ahora se puede indicar al enumerar la secuencia de estados y transiciones en el diagrama que se utiliza en el proceso de reconocimiento. Por ejemplo, el proceso de reconocer **`xtemp`** como un identificador se puede indicar como sigue:

$$\rightarrow 1 \xrightarrow{\texttt{x}} 2 \xrightarrow{\texttt{t}} 2 \xrightarrow{\texttt{e}} 2 \xrightarrow{\texttt{m}} 2 \xrightarrow{\texttt{p}} 2$$

En este diagrama etiquetamos cada transición mediante la letra que se iguala en cada paso.

2.3.1 Definición de los autómatas finitos deterministicos

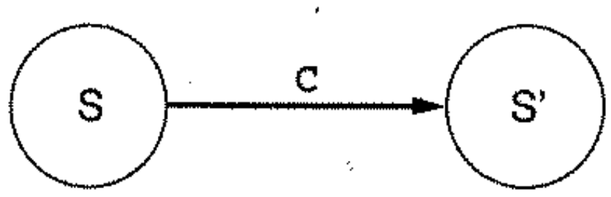
Los diagramas como el que analizamos son descripciones útiles de los autómatas finitos porque nos permiten visualizar fácilmente las acciones del algoritmo. Sin embargo, en ocasiones es necesario tener una descripción más formal de un autómata finito, y por ello procederemos ahora a proporcionar una definición matemática. La mayor parte del tiempo, no obstante, no necesitaremos una visión tan abstracta como ésta, y describiremos la mayoría de los ejemplos en términos del diagrama solo. Otras descripciones de autómatas finitos también son posibles, particularmente las tablas, y éstas serán útiles para convertir los algoritmos en código de trabajo. Las describiremos a medida que surja la necesidad de utilizarlas.

También deberíamos advertir que lo que hemos estado describiendo son autómatas finitos **determinísticos**: autómatas donde el estado siguiente está dado unívocamente por el estado actual y el carácter de entrada actual. Una generalización útil de esto es el **autómata finito no determinístico**, el cual estudiaremos más adelante en esta sección.

Definición

Un **DFA** (por las siglas del concepto autómata finito determinístico en inglés) M se compone de un alfabeto Σ, un conjunto de estados S, una función de transición T: $S \times \Sigma \rightarrow S$, un estado de inicio $s_0 \in S$ y un conjunto de estados de aceptación $A \subset S$. El lenguaje aceptado por M, escrito como $L(M)$, se define como el conjunto de cadenas de caracteres $c_1c_2 \ldots c_n$ con cada $c_i \in \Sigma$, tal que existen estados $s_1 = T(s_0, c_1)$, $s_2 = T(s_1, c_2)$, . . . , $s_n = T(s_{n-1}, c_n)$, con s_n como un elemento de A (es decir, un estado de aceptación).

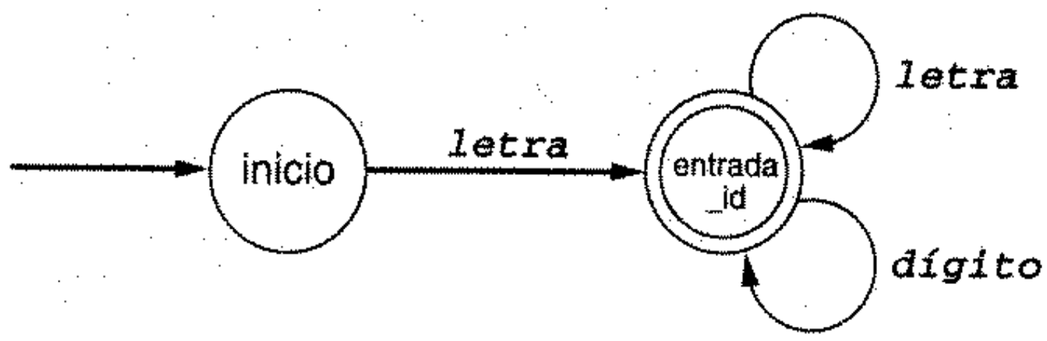
Hacemos las anotaciones siguientes respecto a esta definición. $S \times \Sigma$ se refiere al producto cartesiano o producto cruz de S y Σ: el conjunto de pares (s, c), donde $s \in S$ y $c \in \Sigma$. La función T registra las transiciones: $T(s, c) = s'$ si existe una transición del estado s al estado s' etiquetado mediante c. El segmento correspondiente del diagrama para M tendrá el aspecto siguiente:



La aceptación como la existencia de una secuencia de estados $s_1 = T(s_0, c_1)$, $s_2 = T(s_1, c_2)$, . . . , $s_n = T(s_{n-1}, c_n)$, con s_n siendo un estado de aceptación, significa entonces lo mismo que el diagrama

$$\rightarrow s_0 \xrightarrow{c_1} s_1 \xrightarrow{c_2} s_2 \longrightarrow \cdots \longrightarrow s_{n-1} \xrightarrow{c_n} s_n$$

Advertimos un número de diferencias entre la definición de un DFA y el diagrama del ejemplo identificador. En primer lugar, utilizamos los números para los estados en el diagrama del identificador, mientras la definición no restrinja el conjunto de estados a números. En realidad, podemos emplear cualquier sistema de identificación que queramos para los estados, incluyendo nombres. Por ejemplo, podemos escribir un diagrama equivalente al de la figura 2.1 como



donde ahora denominamos a los estados inicio (porque es el estado de inicio) y entrada_id (porque vimos una letra y estará reconociendo un identificador después de letras y números subsiguientes cualesquiera). El conjunto de estados para este diagrama se convierte ahora en {inicio, entrada_id} en lugar de {1, 2}.

Una segunda diferencia entre el diagrama y la definición es que no etiquetamos las transiciones con caracteres sino con nombres que representan un conjunto de caracteres.

Por ejemplo, el nombre ***letra*** representa cualquier letra del alfabeto de acuerdo con la siguiente definición regular:

```
letra = [a-zA-Z]
```

Ésta es una extensión conveniente de la definición, ya que sería abrumador dibujar 52 transiciones por separado, una para cada letra minúscula y una para cada letra mayúscula. Continuaremos empleando esta extensión de la definición en el resto del capítulo.

Una tercera diferencia fundamental entre la definición y nuestro diagrama es que la definición representa las transiciones como una *función* T: $S \times \Sigma \rightarrow S$. Esto significa que $T(s, c)$ debe tener un valor *para cada s y c*. Pero en el diagrama tenemos T(inicio, c) definida sólo si c es una letra, y T(entrada_id, c) está definida sólo si c es una letra o un dígito. ¿Dónde están las transiciones perdidas? La respuesta es que representan errores; es decir, en el reconocimiento de un identificador no podemos aceptar cualquier otro carácter aparte de letras del estado de inicio y letras o números después de éste.[4] La convención es que estas **transiciones de error** no se dibujan en el diagrama sino que simplemente se supone que siempre existen. Si las dibujáramos, el diagrama para un identificador tendría el aspecto que se ilustra en la figura 2.2.

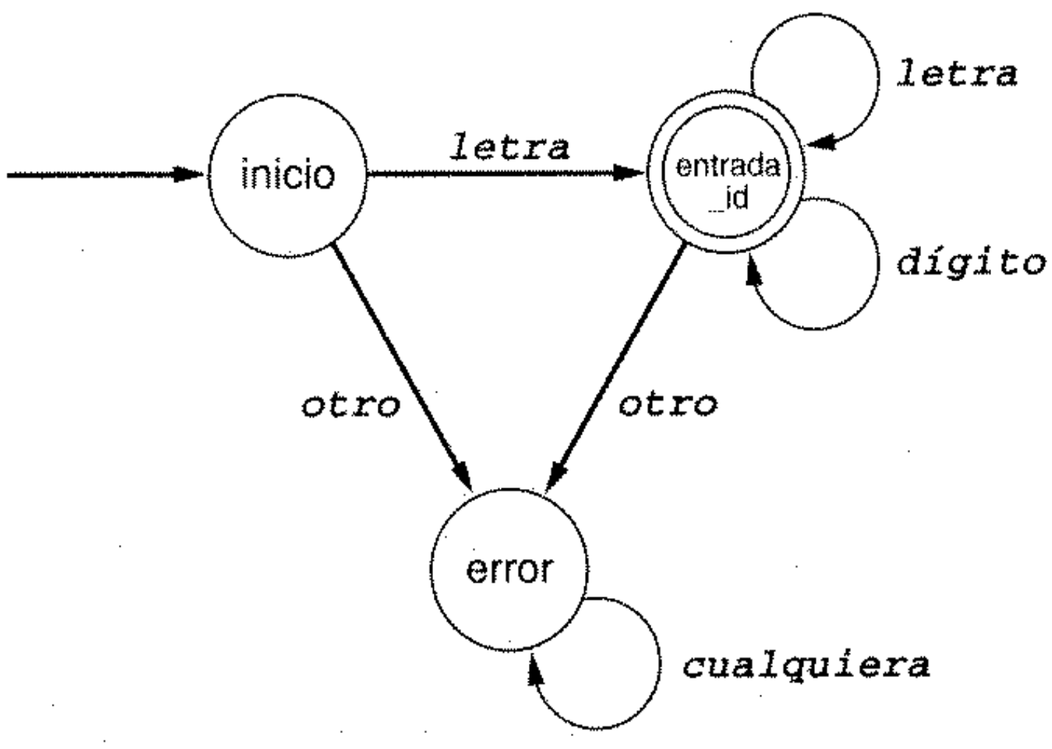


Figura 2.2
Un autómata finito para identificadores con transiciones de error

En esta figura etiquetamos el nuevo estado **error** (porque representa una ocurrencia errónea), y etiquetamos las transiciones de error como ***otro***. Por convención, ***otro*** representa cualquier carácter que no aparezca en ninguna otra transición desde el estado donde se origina. Por consiguiente, la definición de ***otro*** proveniente desde el estado de inicio es

```
otro = ~letra
```

y la definición de ***otro*** proveniente desde el estado **entrada_id** es

```
otro = ~(letra|dígito)
```

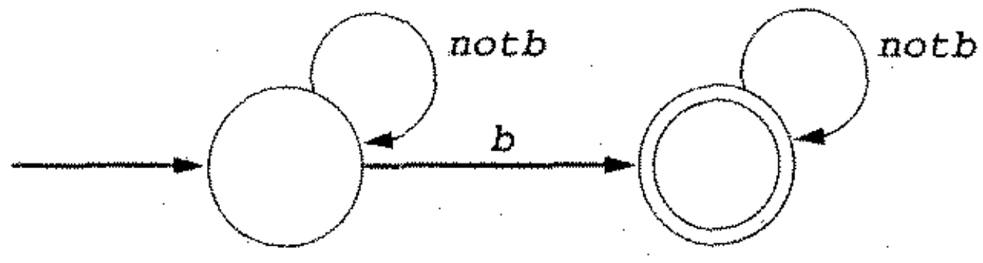
---

4. En realidad, estos caracteres no alfanuméricos significan que no hay un identificador en todo (si estamos en el estado de inicio), o que no hemos encontrado un delimitador que finalice el reconocimiento de un identificador (si estamos en un estado de aceptación). Más adelante en esta sección veremos cómo manejar estos casos.

Advierta también que todas las transiciones desde el estado de error van de regreso hacia sí mismo (etiquetamos a estas transiciones como ***cualquiera*** para indicar que cualquier carácter resulta en esta transición). El estado de error también es de no aceptación. De manera que, una vez que ha ocurrido un error, no podemos escapar del estado de error, y nunca aceptaremos la cadena.

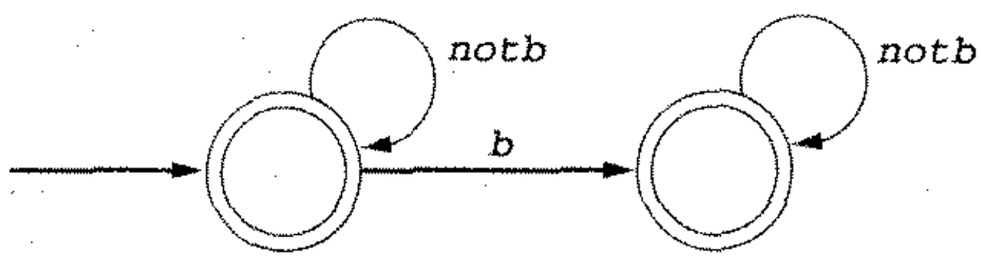
Ahora volveremos a una serie de ejemplos sobre los DFA, en forma paralela a los ejemplos de la sección anterior.

Ejemplo 2.6 El conjunto de cadenas que contienen exactamente una *b* es aceptado por el siguiente DFA:



Advierta que no nos hemos preocupado por etiquetar los estados. Omitiremos las etiquetas cuando no sea necesario referirse a los estados por nombre. §

Ejemplo 2.7 El conjunto de cadenas que contienen como máximo una *b* es aceptado por el siguiente DFA:



Advierta cómo este DFA es una modificación del DFA del ejemplo anterior, obtenido al crear el estado de inicio como un segundo estado de aceptación. §

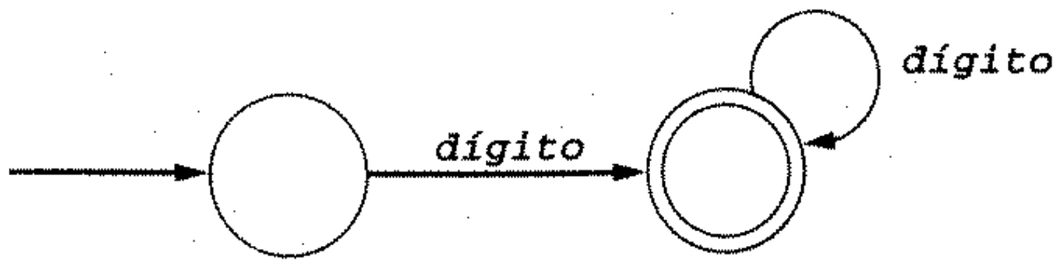
Ejemplo 2.8 En la sección anterior proporcionamos definiciones regulares para constantes numéricas en notación científica como se muestra a continuación:

```
nat = [0-9]+
natconSigno = (+|-)? nat
número = natconSigno("." nat)?(E natconSigno)?
```

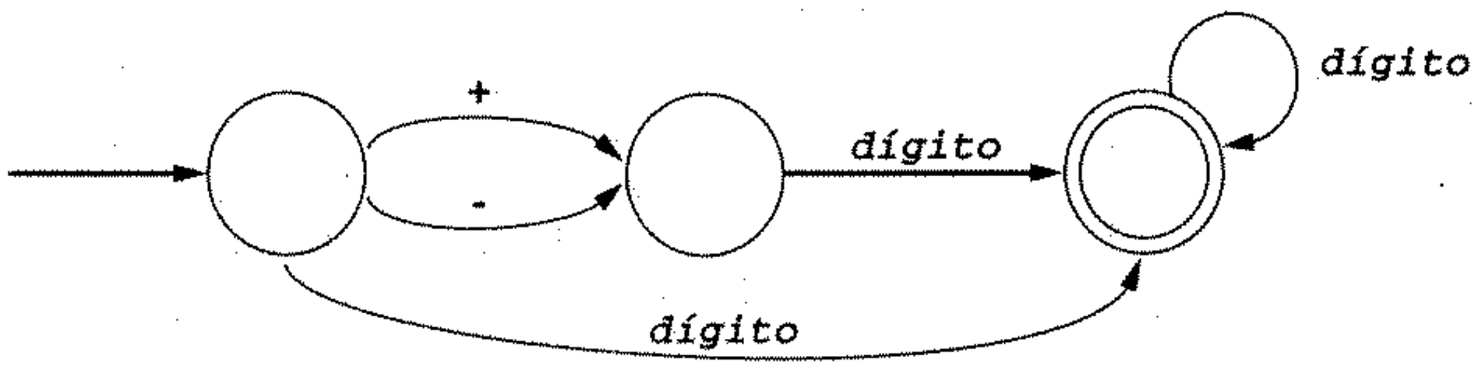
Nos gustaría escribir DFA para las cadenas coincidentes con estas definiciones, pero es útil volverlas a escribir primero como se ve en seguida:

```
dígito = [0-9]
nat = dígito+
natconSigno = (+|-)? nat
número = natconSigno("." nat)?(E natconSigno)?
```

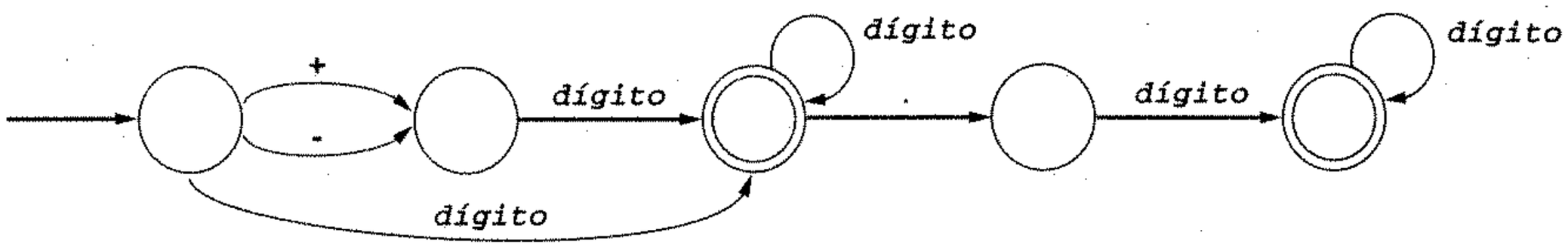
Es sencillo escribir un DFA para ***nat*** como sigue (recuerde que **a+ = aa*** para cualquier **a**):



Un ***naturalconSigno*** es un poco más difícil debido al signo opcional. Sin embargo, podemos observar que un ***naturalconSigno*** comienza con un dígito o con un signo y un dígito, y entonces escribir el siguiente DFA:



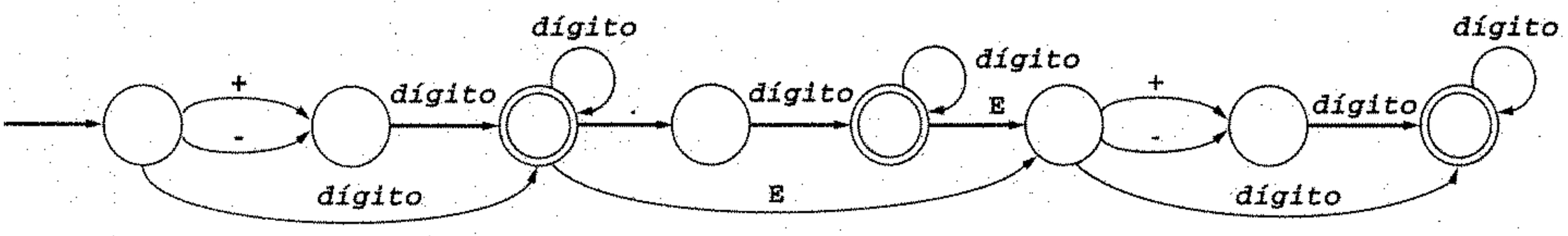
También es fácil agregar la parte fraccional opcional como se aprecia a continuación:



Observe que conservamos ambos estados de aceptación para reflejar el hecho de que la parte fraccional es opcional.

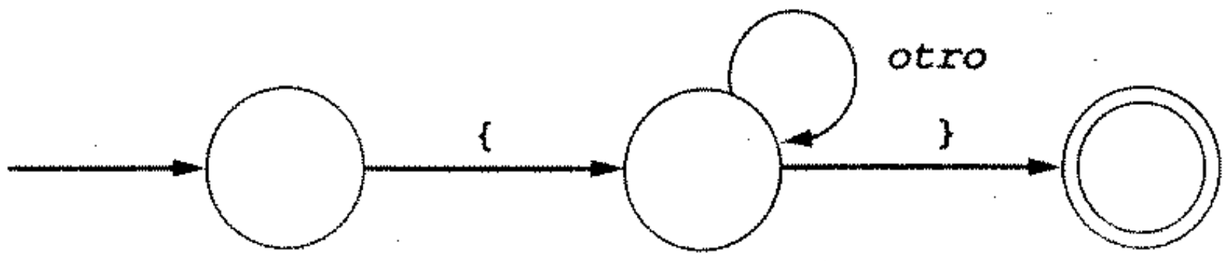
Por último, necesitamos agregar la parte exponencial opcional. Para hacer esto observamos que la parte exponencial debe comenzar con la letra *E* y puede presentarse sólo después de que hayamos alcanzado cualquiera de los estados de aceptación anteriores. El diagrama final se ofrece en la figura 2.3.

Figura 2.3 Un autómata finito para números de punto flotante



§

Ejemplo 2.9 Los comentarios no anidados se pueden describir utilizando DFA. Por ejemplo, los comentarios encerrados entre llaves son aceptados por el siguiente DFA:



En este caso, `otro` significa todos los caracteres, excepto el de la llave derecha. Este DFA corresponde a la expresión regular `{(~})*}`, el cual escribimos previamente en la sección 2.2.4.

Observamos en esa sección que era difícil escribir una expresión regular para comentarios que estuvieran delimitados mediante una secuencia de dos caracteres, como ocurre con los comentarios en C, que son de la forma `/*...(no */s)...*/`. En realidad es más fácil escribir un DFA que acepte tales comentarios que escribir una expresión regular para ellos. Un DFA para esos comentarios en C se proporciona en la figura 2.4.

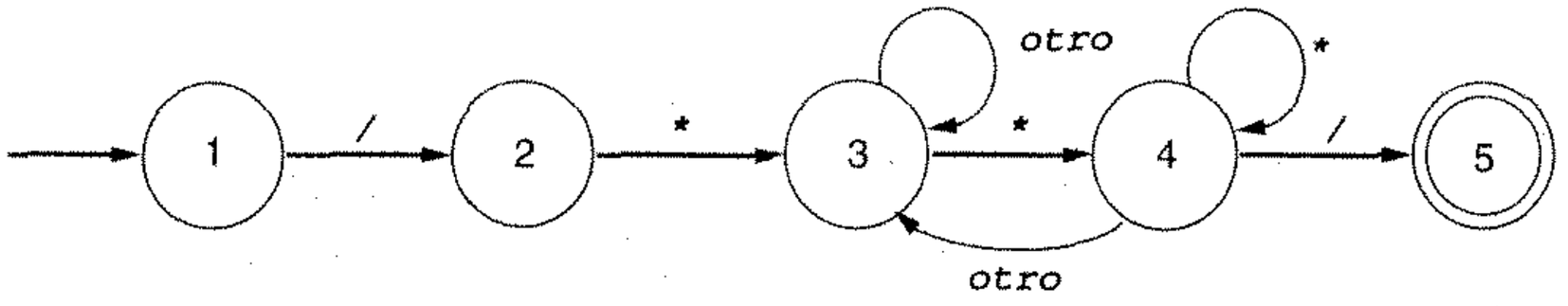


Figura 2.4
Un autómata finito para comentarios tipo C

En esta figura la transición `otro` desde el estado 3 hacia sí mismo se permite para todos los caracteres, excepto el asterisco *, mientras que la transición `otro` del estado 4 al estado 3 se permite para todos los caracteres, excepto el "asterisco" * y la "diagonal" /. Numeramos los estados en este diagrama por simplicidad, pero podríamos haberles dado nombres más significativos, como los siguientes (con los números correspondientes entre paréntesis): inicio (1); preparar_comentario (2); entrada_comentario (3); salida_comentario (4) y final (5). §

2.3.2 Búsqueda hacia delante, retroseguimiento y autómatas no determinísticos

Estudiamos los DFA como una manera de representar algoritmos que aceptan cadenas de caracteres de acuerdo con un patrón. Como tal vez el lector ya haya adivinado, existe una fuerte relación entre una expresión regular para un patrón y un DFA que acepta cadenas de acuerdo con el patrón. Investigaremos esta relación en la siguiente sección. Pero primero necesitamos estudiar más detalladamente los algoritmos precisos que representan los DFA, porque en algún momento tendremos que convertirlos en el código para un analizador léxico.

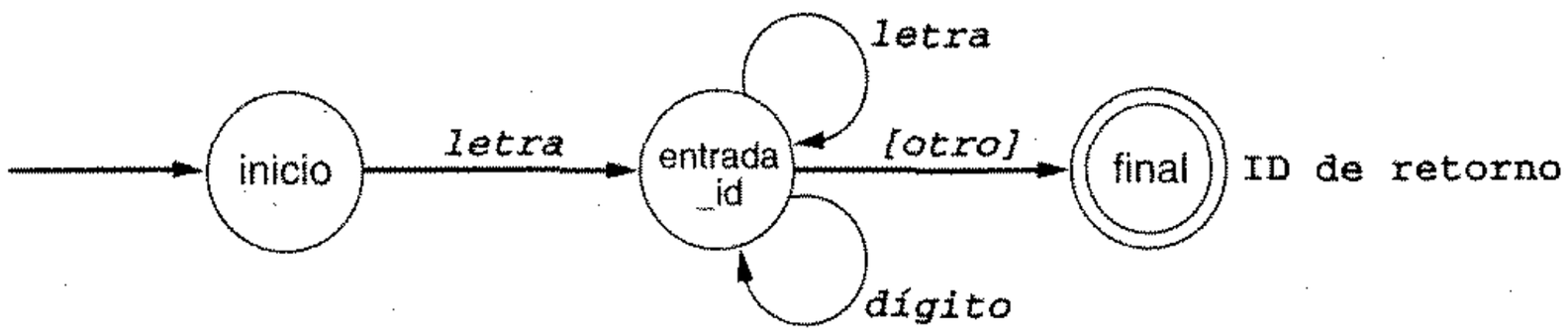
Ya advertimos que el diagrama de un DFA no representa todo lo que necesita un DFA, sino que sólo proporciona un esbozo de su funcionamiento. En realidad, vimos que la definición matemática implica que un DFA debe tener una transición para cada estado y carácter, y que aquellas transiciones que dan errores como resultado por lo regular se dejan fuera del diagrama para el DFA. Pero incluso la definición matemática no describe todos los aspectos del comportamiento de un algoritmo de DFA. Por ejemplo, no especifica lo que ocurre cuando se presenta un error. Tampoco especifica la acción que tomará un programa al alcanzar un estado de aceptación, o incluso cuando iguale un carácter durante una transición.

Una acción típica que ocurre cuando se hace una transición es mover el carácter de la cadena de entrada a una cadena que acumula los caracteres hasta convertirse en un token simple (el valor de cadena del token o lexema del token). Una acción típica cuando se alcanza un estado de aceptación es devolver el token que se acaba de reconocer, junto con cualquier atributo asociado. Una acción típica cuando se alcanza un estado de error es retroceder hacia la entrada (retroseguimiento) o generar un token de error.

Nuestro ejemplo original de un token de identificador muestra gran parte del comportamiento que deseamos describir aquí, y así regresaremos al diagrama de la figura 2.4. El DFA

de esa figura no muestra el comportamiento que queremos de un analizador léxico por varias razones. En primer lugar, el estado de error no es en realidad un error en absoluto, pero representa el hecho de que un identificador no va a ser reconocido (si venimos desde el estado de inicio), o bien, que se ha detectado un delimitador y ahora deberíamos aceptar y generar un token de identificador. Supongamos por el momento (lo que de hecho es el comportamiento correcto) que existen otras transiciones representando las transiciones sin letra desde el estado de inicio. Entonces podemos indicar que se ha detectado un delimitador desde el estado entrada_id, y que debería generarse un token identificador mediante el diagrama de la figura 2.5:

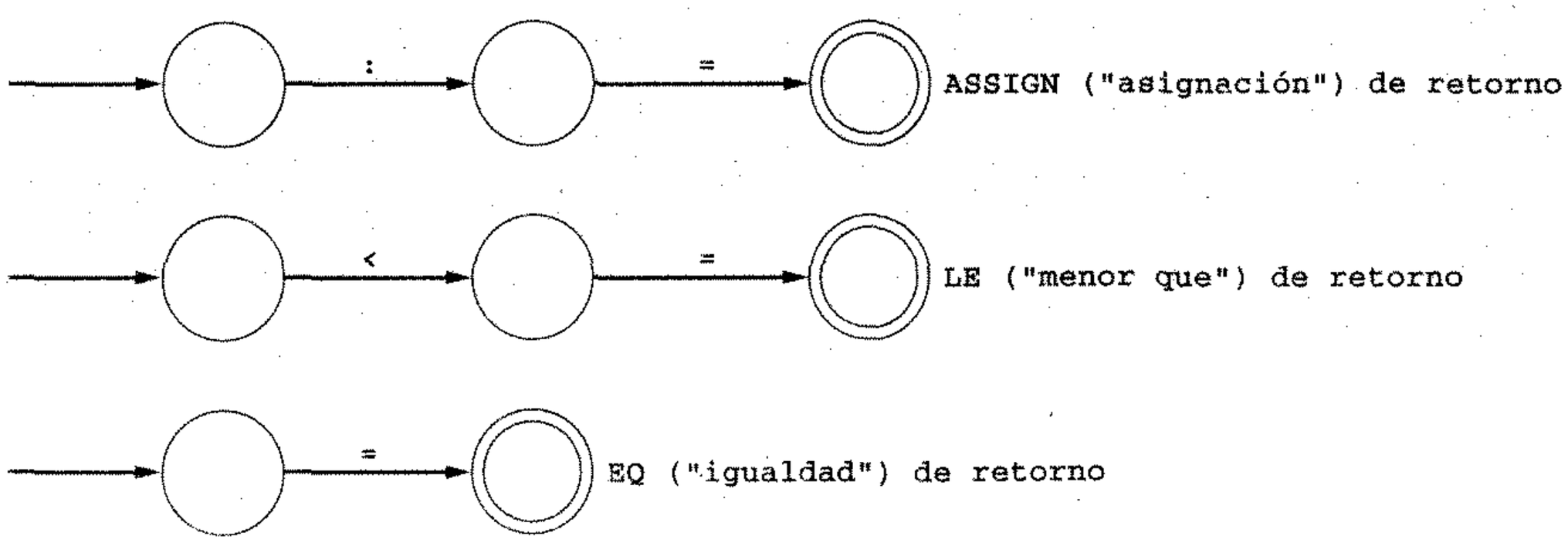
Figura 2.5
Un autómata finito para un identificador con valor de retorno y delimitador



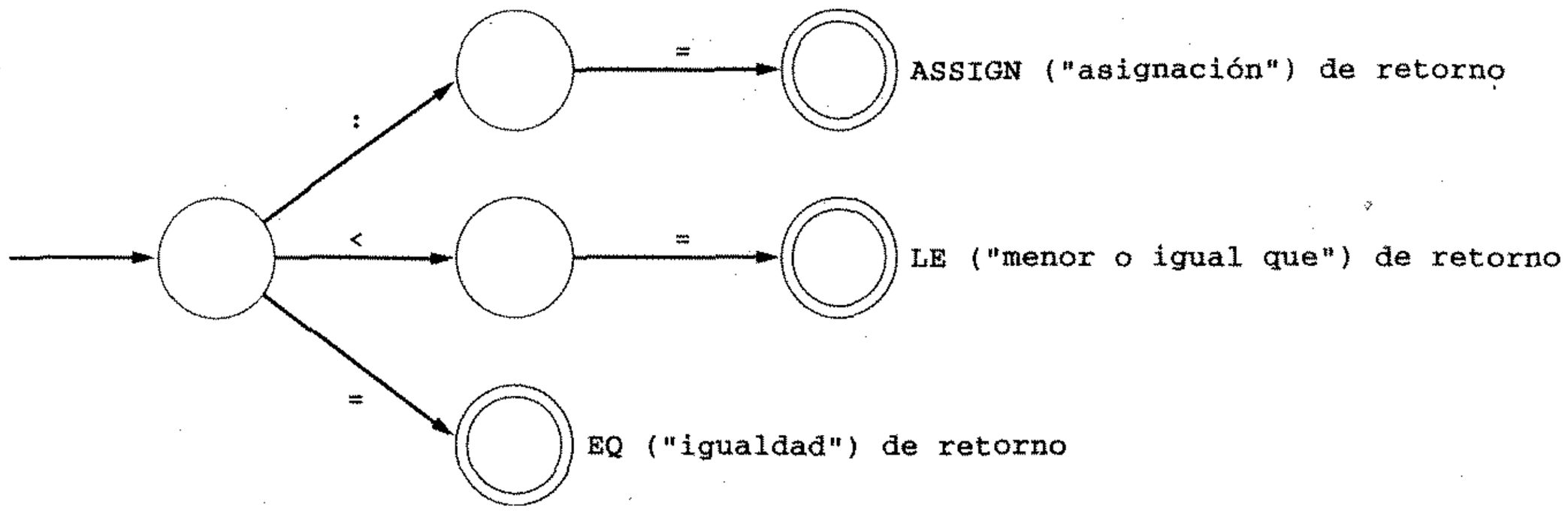
En el diagrama encerramos la transición *otro* entre corchetes para indicar que el carácter delimitador se debería considerar como búsqueda hacia delante, es decir, que debería ser devuelto a la cadena de entrada y no consumido. Además, el estado de error se ha convertido en el estado de aceptación en este diagrama y no hay transiciones fuera del estado de aceptación. Esto es lo que queremos, puesto que el analizador léxico debería reconocer un token a la vez y comenzar de nuevo en su estado de inicio después de reconocer cada token.

Este nuevo diagrama también expresa el principio de la subcadena más larga descrito en la sección 2.2.4: el DFA continúa igualando letras y dígitos (en estado entrada_id) hasta que se encuentra un delimitador. En contraste, el diagrama antiguo permitiría aceptar al DFA en cualquier punto mientras se lee una cadena de identificador, algo que desde luego no queremos que ocurra.

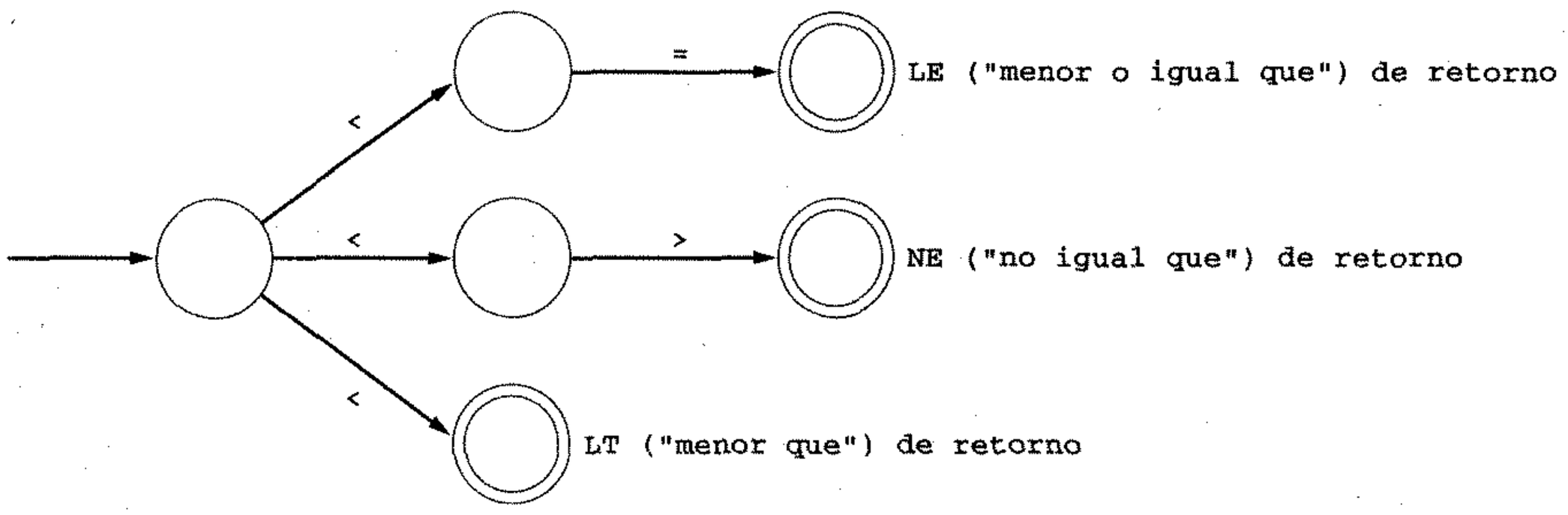
Ahora volvamos nuestra atención a la cuestión de cómo llegar al estado de inicio en primer lugar. En un lenguaje de programación típico existen muchos tokens, y cada token será reconocido por su propio DFA. Si cada uno de estos tokens comienza con un carácter diferente, entonces es fácil conjuntarlos uniendo simplemente todos sus estados de inicio en un solo estado de inicio. Por ejemplo, considere los tokens dados por las cadenas :=, <= e =. Cada uno de éstos es una cadena fija, y los DFA para ellos se pueden escribir como sigue:



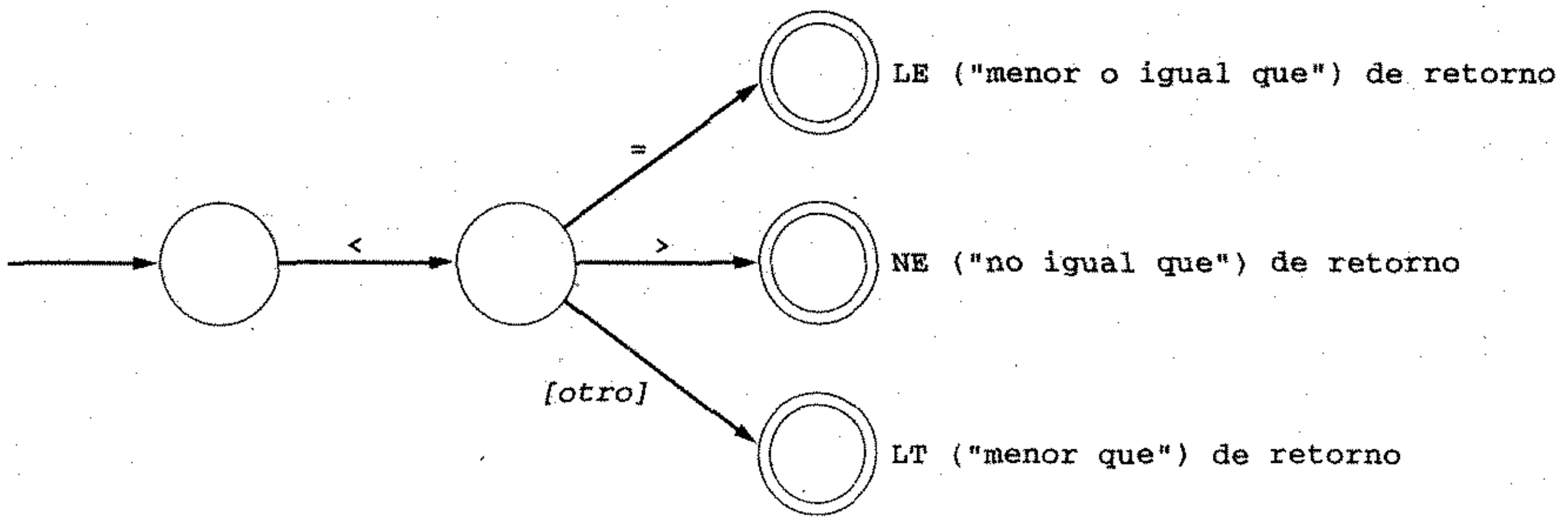
Como cada uno de estos tokens comienza con un carácter diferente, podemos sólo identificar sus estados de inicio para obtener el siguiente DFA:



Sin embargo, supongamos que teníamos varios tokens que comenzaban con el mismo carácter, tales como <, <= y <>. Ahora no podemos limitarnos a escribir el diagrama siguiente, puesto que no es un DFA (dado un estado y un carácter, siempre debe haber una transición única hacia un nuevo estado único):



En vez de eso debemos arreglarlo de manera que sólo quede una transición por hacerse en cada estado, tal como en el diagrama siguiente:

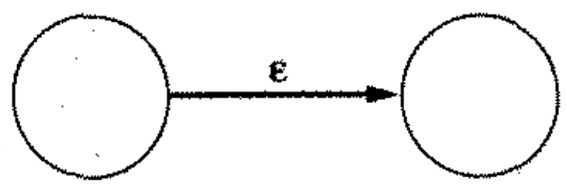


En principio, deberíamos poder combinar todos los tokens en un DFA gigante de esta manera. No obstante, la complejidad de una tarea así se vuelve enorme, especialmente si se hace de una manera no sistemática.

Una solución a este problema es ampliar la definición de un autómata finito para incluir el caso en el que pueda existir más de una transición para un carácter particular, mientras que al mismo tiempo se desarrolla un algoritmo para convertir sistemáticamente estos nuevos autómatas finitos generalizados en DFA. Aquí describiremos estos autómatas generalizados, mientras que pospondremos la descripción del algoritmo de traducción hasta la siguiente sección.

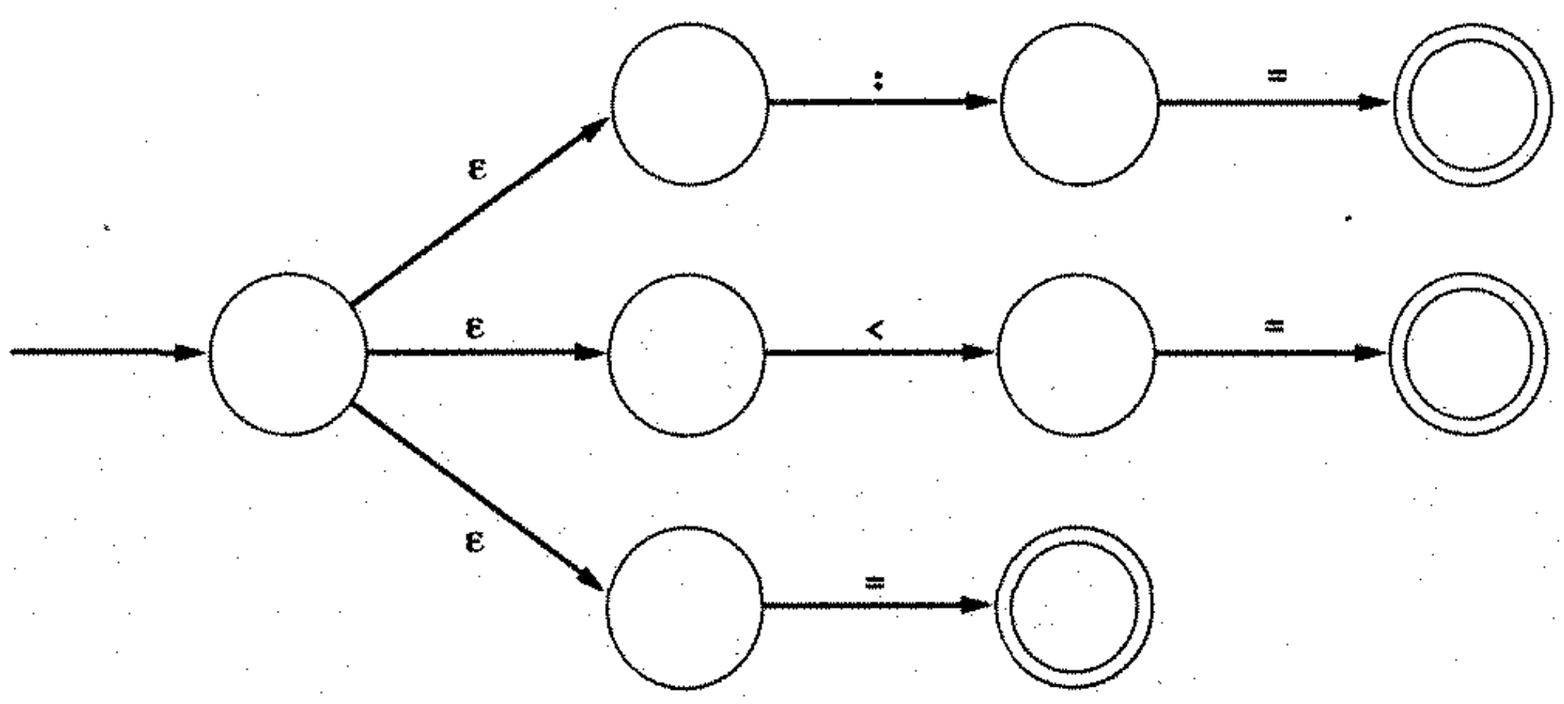
La nueva clase de autómatas finitos se denomina **NFA** por las siglas del término **autómata finito no determinístico** en inglés. Antes de definirlo necesitamos otra generalización que será útil en la aplicación de los autómatas finitos a los analizadores léxicos: el concepto de la transición ε.

Una **transición ε** es una transición que puede ocurrir sin consultar la cadena de entrada (y sin consumir ningún carácter). Se puede ver como una "igualación" de la cadena vacía, la cual antes describimos como ε. En un diagrama una transición ε se describe como si ε fuera en realidad un carácter:

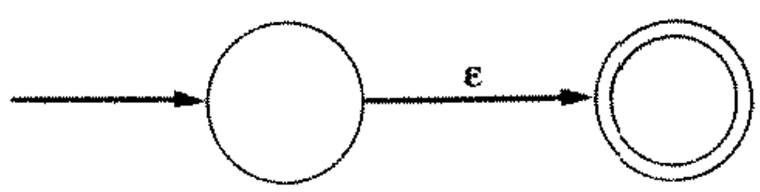


Esto no debe confundirse con una correspondencia del carácter ε en la entrada: si el alfabeto incluye este carácter, se debe distinguir del carácter ε que se usa como metacarácter para representar una transición ε.

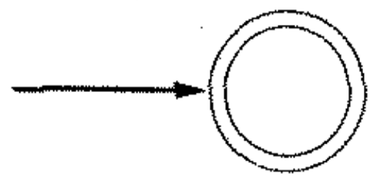
Las transiciones ε son hasta cierto punto contraintuitivas, porque pueden ocurrir "espontáneamente", es decir, sin búsqueda hacia delante y sin modificación de la cadena de entrada, pero son útiles de dos maneras. Una es porque permiten expresar una selección de alternativas de una manera que no involucre la combinación de estados. Por ejemplo, la selección de los tokens :=, <= e = se puede expresar al combinar los autómatas para cada token como se muestra a continuación:



Esto tiene la ventaja de que mantiene intacto al autómata original y sólo agrega un nuevo estado de inicio para conectarlos. La segunda ventaja de las transiciones ε es que pueden describir explícitamente una coincidencia de la cadena vacía:

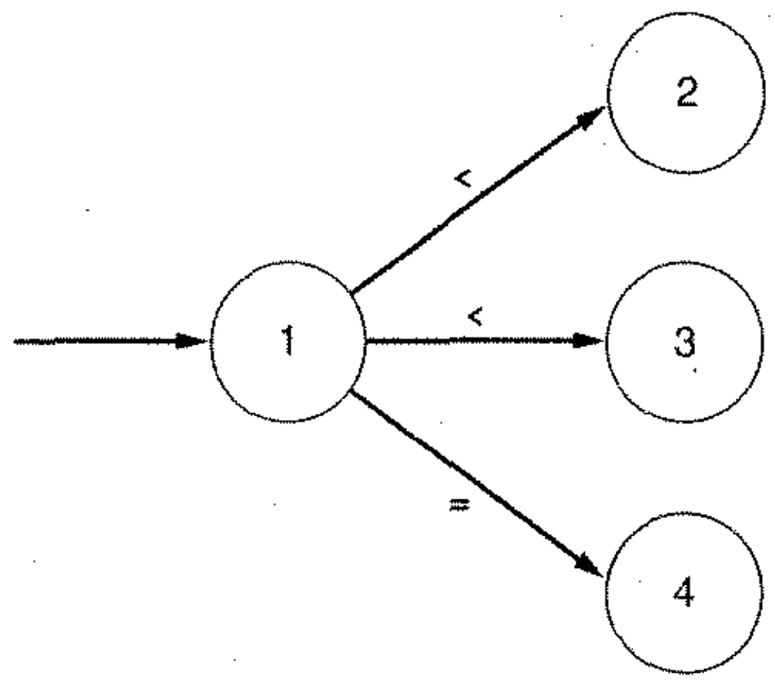


Por supuesto, esto es equivalente al siguiente DFA, el cual expresa que la aceptación ocurriría sin hacer coincidir ningún carácter:



Pero es útil tener la notación explícita anterior.

Ahora definiremos qué es un autómata no determinístico, el cual tiene una definición muy similar a la de un DFA, pero en este caso, según el análisis anterior, necesitamos ampliar el alfabeto Σ para incluir a ε. Haremos esto escribiendo $\Sigma \cup \{\varepsilon\}$ (la unión de Σ y ε), donde antes utilizamos Σ (esto supone que ε no es originalmente un miembro de Σ). También necesitamos ampliar la definición de T (la función de transición) de modo que cada carácter pueda conducir a más de un estado. Hacemos esto permitiendo que el valor de T sea un *conjunto* de estados en lugar de un estado simple. Por ejemplo, dado el diagrama



tenemos $T(1, <) = \{2, 3\}$. En otras palabras, desde el estado 1 podemos moverlos a cualesquiera de los estados 2 o 3 en el carácter de entrada $<$, y T se convierte en una función que mapea pares estado/símbolo hacia *conjuntos de estados*. Por consiguiente, el rango de T es el **conjunto de potencia** del conjunto S de estados (el conjunto de todos los subconjuntos de S); escribimos esto como $\wp(S)$ (p manuscrita de S). Ahora estableceremos la definición.

Definición

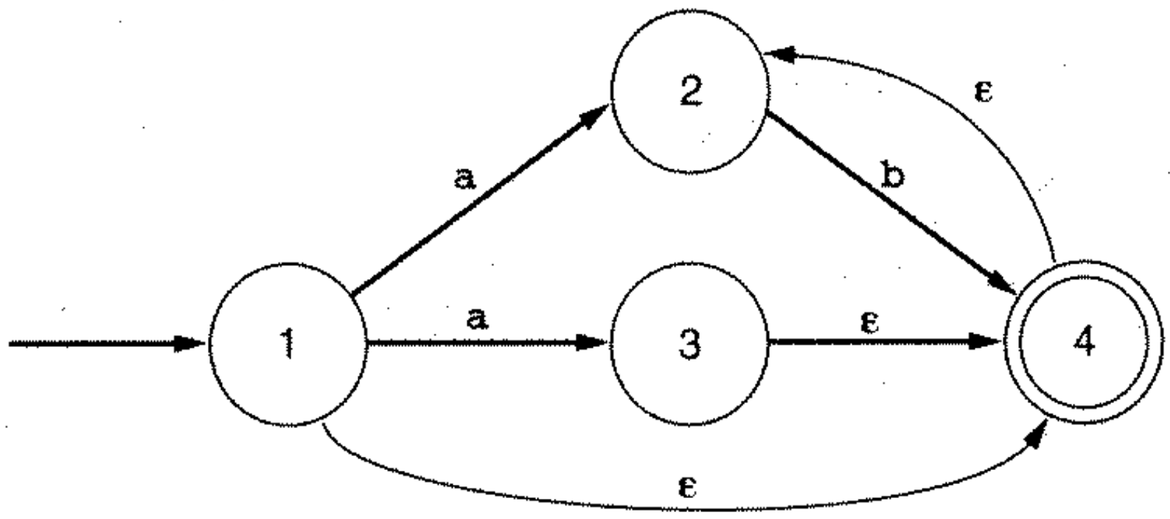
Un **NFA** (por las siglas del término autómata finito no determinístico en inglés) M consta de un alfabeto Σ, un conjunto de estados S y una función de transición T: $S \times (\Sigma \cup \{\varepsilon\}) \rightarrow \wp(S)$, así como de un estado de inicio s_0 de S y un conjunto de estados de aceptación A de S. El lenguaje aceptado por M, escrito como $L(M)$, se define como el conjunto de cadenas de caracteres $c_1c_2 \ldots c_n$ con cada c_i de $\Sigma \cup \{\varepsilon\}$ tal que existen estados s_1 en $T(s_0, c_1)$, s_2 en $T(s_1, c_2)$, . . . , s_n en $T(s_{n-1}, c_n)$, con s_n como un elemento de A.

De nuevo necesitamos advertir algunas cosas sobre esta definición. Cualquiera de las c_i en $c_1c_2 \ldots c_n$ puede ser ε, y la cadena que en realidad es aceptada es la cadena $c_1c_2 \ldots c_n$ con la ε eliminada (porque la concatenación de s con ε es s misma). Por lo tanto, la cadena

$c_1c_2 \ldots c_n$ puede en realidad contener menos de n caracteres. Además, la secuencia de estados $s_1, \ldots, s_n$ se elige de los *conjuntos* de estados $T(s_0, c_1), \ldots, T(s_{n-1}, c_n)$, y esta elección no siempre estará determinada de manera única. Ésta, de hecho, es la razón por la que los autómatas se denominan *no determinísticos*: la secuencia de transiciones que acepta una cadena particular no está determinada en cada paso por el estado y el siguiente carácter de entrada. En realidad, pueden introducirse números arbitrarios de ε en cualquier punto de la cadena, los cuales corresponden a cualquier número de transiciones ε en la NFA. De este modo, una NFA no representa un algoritmo. Sin embargo, se puede simular mediante un algoritmo que haga un retroseguimiento a través de cada elección no determinística, como veremos más adelante en esta sección.

No obstante, consideremos primero un par de ejemplos de NFA.

Ejemplo 2.10 Considere el siguiente diagrama de un NFA.

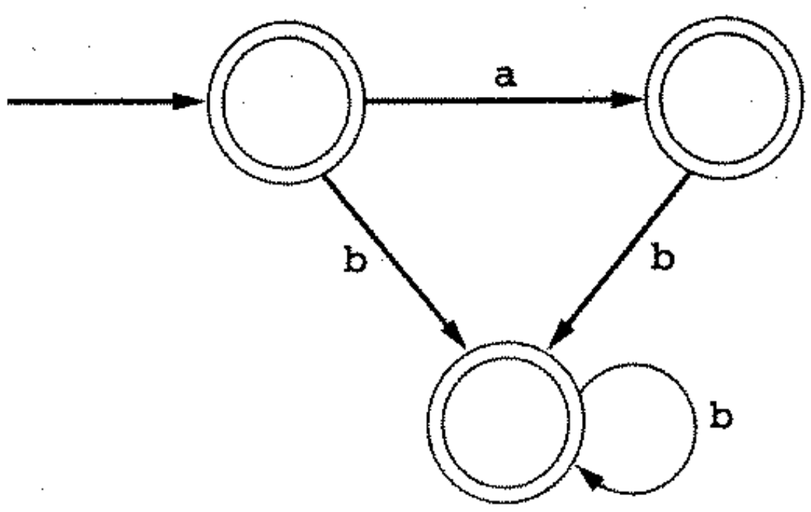


La cadena **abb** puede ser aceptada por cualquiera de las siguientes secuencias de transiciones:

$$\rightarrow 1 \xrightarrow{\texttt{a}} 2 \xrightarrow{\texttt{b}} 4 \xrightarrow{\varepsilon} 2 \xrightarrow{\texttt{b}} 4$$

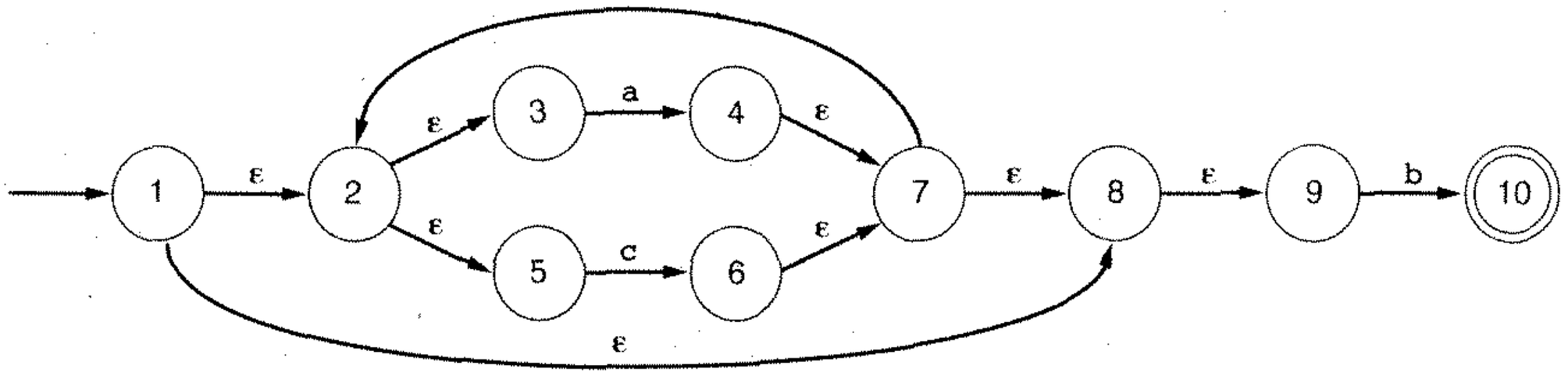
$$\rightarrow 1 \xrightarrow{\texttt{a}} 3 \xrightarrow{\varepsilon} 4 \xrightarrow{\varepsilon} 2 \xrightarrow{\texttt{b}} 4 \xrightarrow{\varepsilon} 2 \xrightarrow{\texttt{b}} 4$$

En realidad las transiciones del estado 1 al estado 2 en *a*, y del estado 2 al estado 4 en *b*, permiten que la máquina acepte la cadena *ab*, y entonces, utilizando la transición ε del estado 4 al estado 2, todas las cadenas igualan la expresión regular **ab+**. De manera similar, las transiciones del estado 1 al estado 3 en *a*, y del estado 3 al estado 4 en ε, permiten la aceptación de todas las cadenas que coinciden con **ab***. Finalmente, siguiendo la transición ε desde el estado 1 hasta el estado 4 se permite la aceptación de todas las cadenas coincidentes con **b***. De este modo, este NFA acepta el mismo lenguaje que la expresión regular **ab+|ab*|b***. Una expresión regular más simple que genera el mismo lenguaje es **(a|ε)b***. El siguiente DFA también acepta este lenguaje:

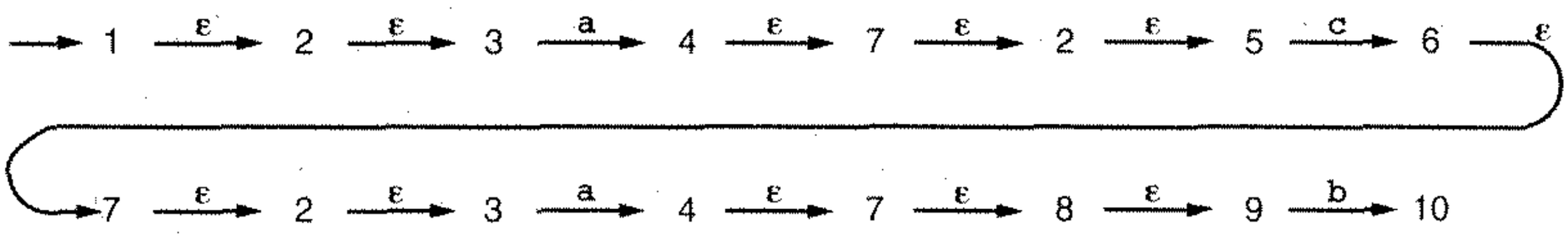


§

Ejemplo 2.11 Considere el siguiente NFA:



Éste acepta la cadena *acab* al efectuar las transiciones siguientes:

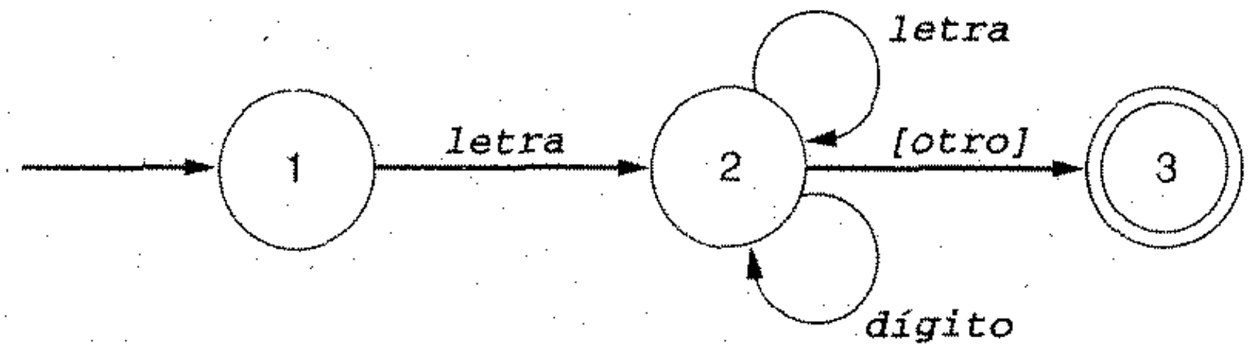


De hecho, no es difícil ver que este NFA acepta el mismo lenguaje que el generado por la expresión regular `(a|c)b*`. §

2.3.3 Implementación de autómatas finitos en código

Existen diversas maneras de traducir un DFA o un NFA en código, y las analizaremos en esta sección. Sin embargo, no todos estos métodos serán útiles para un analizador léxico de compilador, y en las últimas dos secciones de este capítulo se demostrarán los aspectos de la codificación apropiada para analizadores léxicos con más detalle.

Considere de nueva cuenta nuestro ejemplo original de un DFA que acepta identificadores compuestos de una letra seguida por una secuencia de letras y/o dígitos, en su forma corregida que incluye búsqueda hacia delante y el principio de la subcadena más larga (véase la figura 2.5):



La primera y más sencilla manera de simular este DFA es escribir el código de la manera siguiente:

```
{ iniciando en el estado 1 }
if el siguiente carácter es una letra then
  avanzan en la entrada;
  { ahora en el estado 2 }
  while el siguiente carácter es una letra o un dígito do
```

(continúa)

```
    avanza en la entrada; { permanece en el estado 2 }
  end while;
  { ir al estado 3 sin avanzar en la entrada }
  aceptar;
else
  { error u otros casos }
end if;
```

Tales códigos utilizan la posición en el código (anidado dentro de pruebas) para mantener el estado implícitamente, como indicamos mediante los comentarios. Esto es razonable si no hay demasiados estados (lo que requeriría muchos niveles de anidación), y si las iteraciones en el DFA son pequeños. Un código como éste se ha utilizado para escribir pequeños analizadores léxicos. Pero existen dos desventajas con este método. La primera es que es de propósito específico, es decir, cada DFA se tiene que tratar de manera un poco diferente, y es difícil establecer un algoritmo que traduzca cada DFA a código de esta manera. La segunda es que la complejidad del código aumenta de manera dramática a medida que el número de estados se eleva o, más específicamente, a medida que se incrementa el número de estados diferentes a lo largo de trayectorias arbitrarias. Como un ejemplo simple de estos problemas, consideremos el DFA del ejemplo 2.9 como se dio en la figura 2.4 (página 53) que acepta comentarios en C, el cual se podría implementar por medio de código de la forma siguiente:

```
{ estado 1 }
if el siguiente carácter es "/" then
  avanzan en la entrada; { estado 2 }
  if el siguiente carácter es "*" then
    adelantar la entrada; { estado 3 }
    hecho := false;
    while not hecho do
      while el siguiente carácter de entrada no es "*" do
        avanza en la entrada;
      end while;
      avanza en la entrada; { estado 4 }
      while el siguiente carácter de entrada es "*" do
        avanza en la entrada;
      end while;
      if el siguiente carácter de entrada es "/" then
        hecho := true;
      end if;
      avanza en la entrada;
    end while;
    aceptar; { estado 5 }
  else { otro procesamiento }
  end if;
else { otro procesamiento }
end if;
```

Advierta el considerable incremento en complejidad, y la necesidad de lidiar con la iteración que involucra los estados 3 y 4 mediante el uso de la variable booleana *hecho*.

Un método de implementación sustancialmente mejor se obtiene al utilizar una variable para mantener el estado actual y escribir las transiciones como una sentencia case doblemente anidada dentro de una iteración, donde la primera sentencia case prueba el estado actual y el segundo nivel anidado prueba el carácter de entrada, lo que da el estado. Por ejemplo, el DFA anterior para identificadores se puede traducir en el esquema de código de la figura 2.6.

Figura 2.6
Implementación del identificador DFA utilizando una variable de estado y pruebas case anidadas

```
estado := 1; { inicio }
while estado = 1 o 2 do
  case estado of
  1:  case carácter de entrada of
      letra: avanza en la entrada;
                  estado := 2;
      else estado := . . . { error u otro };
      end case;
  2:  case carácter de entrada of
      letra, dígito: avanza en la entrada;
                  estado := 2; { realmente innecesario }
      else estado := 3;
      end case;
  end case;
end while;
if estado = 3 then aceptar else error ;
```

Observe cómo este código refleja el DFA de manera directa: las transiciones corresponden a la asignación de un nuevo estado a la variable *estado* y avanzan en la entrada (excepto en el caso de la transición "que no consume" del estado 2 al estado 3).

Ahora el DFA para los comentarios en C (figura 2.4) se puede traducir en el esquema de código más legible de la figura 2.7. Una alternativa para esta organización es tener el case exterior basado en el carácter de entrada y los case internos basados en el estado actual (véanse los ejercicios).

En los ejemplos que acabamos de ver, el DFA se "conectó" directamente dentro del código. También es posible expresar el DFA como una estructura de datos y entonces escribir un código "genérico" que tomará sus acciones de la estructura de datos. Una estructura de datos simple que es adecuada para este propósito es una **tabla de transición**, o arreglo bidimensional, indizado por estado y carácter de entrada que expresa los valores de la función de transición T:

	Caracteres en el alfabeto c
Estados s	Estados representando transiciones $T(s, c)$

Como ejemplo, el DFA para identificadores se puede representar como la siguiente tabla de transición:

estado \ carácter de entrada	letra	dígito	otro
1	2		
2	2	2	3
3			

Figura 2.7
Implementación del DFA de la figura 2.4

```
estado := 1; { inicio }
while estado = 1, 2, 3 o 4 do
  case estado of
  1:  case carácter de entrada of
      "/" : avanza en la entrada;
            estado := 2;
      else estado := . . . { error u otro };
      end case;
  2:  case carácter de entrada of
      "*": avanza en la entrada;
            estado := 3;
      else estado := . . . { error u otro };
      end case;
  3:  case carácter de entrada of
      "*": avanza en la entrada;
            estado := 4;
      else advance the input { y permanecer en el estado 3 };
      end case;
  4:  case carácter de entrada of
      "/" avanza en la entrada;
            estado := 5;
      "*": avanza en la entrada; { y permanecer en el estado 4 }
      else avanza en la entrada;
            estado := 3;
      end case;
  end case;
end while;
if estado = 5 then aceptar else error ;
```

En esta tabla las entradas en blanco representan transiciones que no se muestran en el diagrama DFA (es decir, representan transiciones a estados de error u otros procesamientos). También suponemos que el primer estado que se muestra es el estado de inicio. Sin embargo, esta tabla no indica cuáles estados aceptados y cuáles transiciones no consumen sus entradas. Esta información puede conservarse en la misma estructura de datos que representa la tabla o en una estructura de datos por separado. Si agregamos esta información a la tabla de transición anterior (utilizando una columna por separado para indicar estados de aceptación y corchetes para señalar transiciones "no consumidoras de entrada"), obtenemos la tabla siguiente:

carácter de entrada / estado	letra	dígito	otro	Aceptación
1	2			no
2	2	2	[3]	no
3				sí

Como un segundo ejemplo de una tabla de transición, presentamos la tabla para el DFA correspondiente a los comentarios en C (nuestro segundo ejemplo presentado con anterioridad):

estado \ carácter de entrada	/	*	otro	*Aceptación*
1	2			no
2		3		no
3	3	4	3	no
4	5	4	3	no
5				sí

Ahora podemos escribir del código en una forma que implementará cualquier DFA, dadas las entradas y estructuras de datos apropiadas. El siguiente esquema de código supone que las transiciones se mantienen en un arreglo de transición *T* indizado por estados y carácter de entrada; que las transiciones que avanzan en la entrada (es decir, aquellas que no están marcadas con corchetes en la tabla) están dadas por el arreglo booleano *Avanzar*, indizado también por estados y caracteres de entrada; y que los estados de aceptación están dados por el arreglo booleano *Aceptar*, indizado por estados. El siguiente es el esquema de código:

```
estado := 1;
ch := siguiente carácter de entrada;
while not Aceptar[estado] and not error(estado) do
  nuevoestado := T[estado,ch];
  if Avanzar[estado,ch] then ch := siguiente carácter de entrada;
  estado := nuevoestado;
end while;
if Aceptar[estado] then aceptar;
```

Los métodos algorítmicos como los que acabamos de describir se denominan **controlados por tabla** porque emplean tablas para dirigir el progreso del algoritmo. Los métodos controlados por tabla tienen ciertas ventajas: el tamaño del código se reduce, el mismo código funcionará para muchos problemas diferentes y el código es más fácil de modificar (mantener). La desventaja es que las tablas pueden volverse muy grandes y provocar un importante aumento en el espacio utilizado por el programa. En realidad, gran parte del espacio en los arreglos que acabamos de describir se desperdicia. Por lo tanto, los métodos controlados por tabla a menudo dependen de métodos de compresión de tablas, tales como representaciones de arreglos dispersos, aunque por lo regular existe una penalización en tiempo que se debe pagar por dicha compresión, ya que la búsqueda en tablas se vuelve más lenta. Como los analizadores léxicos deben ser eficientes, rara vez se utilizan estos métodos para ellos, aunque se pueden emplear en programas generadores de analizadores léxicos tales como Lex. No los estudiaremos más aquí.

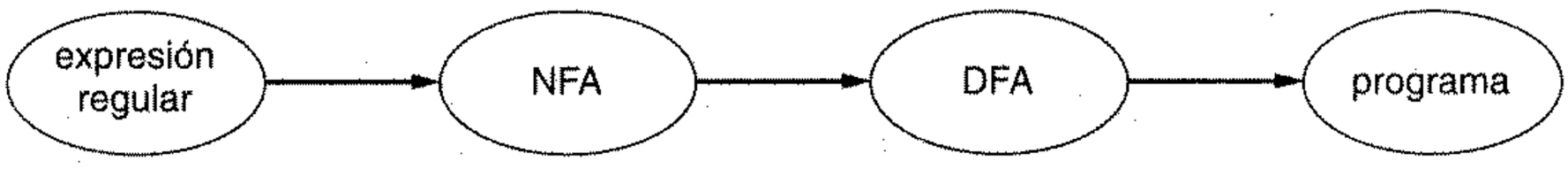
Por último, advertimos que los NFA se pueden implementar de maneras similares a los DFA, excepto que como los NFA son no determinísticos, tienen muchas secuencias diferentes de transiciones en potencia que se deben probar. Por consiguiente, un programa que simula un NFA debe almacenar transiciones que todavía no se han probado y realizar el retroseguimiento a ellas en caso de falla. Esto es muy similar a los algoritmos que intentan encontrar trayectorias en gráficas dirigidas, sólo que la cadena de entrada guía la búsqueda.

Como los algoritmos que realizan una gran cantidad de retroseguimientos tienden a ser poco eficientes, y un analizador léxico debe ser tan eficiente como sea posible, ya no describiremos tales algoritmos. En su lugar, se puede resolver el problema de simular un NFA por medio del método que estudiaremos en la siguiente sección, el cual convierte un NFA en un DFA. Así que pasaremos a esta sección, donde regresaremos brevemente al problema de simular un NFA.

2.4 DESDE LAS EXPRESIONES REGULARES HASTA LOS DFA

En esta sección estudiaremos un algoritmo para traducir una expresión regular en un DFA. También existe un algoritmo para traducir un DFA en una expresión regular, de manera que las dos nociones son equivalentes. Sin embargo, debido a lo compacto de las expresiones regulares, se suele preferir a los DFA como descripciones de token, y de este modo la generación del analizador léxico comienza comúnmente con expresiones regulares y se sigue a través de la construcción de un DFA hasta un programa de analizador léxico final. Por esta razón, nuestro interés se enfocará sólo en un algoritmo que realice esta dirección de la equivalencia.

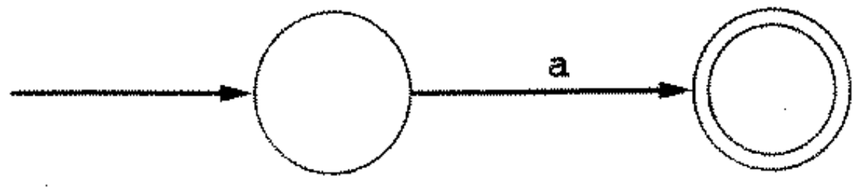
El algoritmo más simple para traducir una expresión regular en un DFA pasa por una construcción intermedia, en la cual se deriva un NFA de la expresión regular, y posteriormente se emplea para construir un DFA equivalente. Existen algoritmos que pueden traducir una expresión regular de manera directa en un DFA, pero son más complejos y la construcción intermedia también es de cierto interés. Así que nos concentraremos en la descripción de dos algoritmos, uno que traduce una expresión regular en un NFA y el segundo que traduce un NFA en un DFA. Combinado con uno de los algoritmos para traducir un DFA en un programa descrito en la sección anterior, el proceso de la construcción de un analizador léxico se puede automatizar en tres pasos, como se ilustra mediante la figura que se ve a continuación:



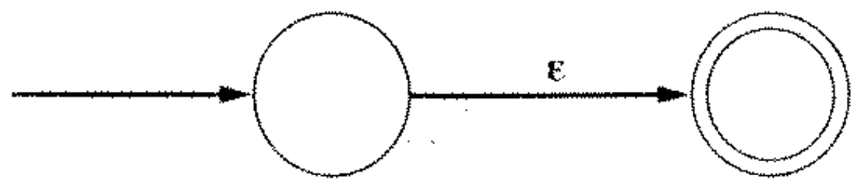
2.4.1 Desde una expresión regular hasta un NFA

La construcción que describiremos se conoce como la **construcción de Thompson**, en honor a su inventor. Utiliza transiciones ε para "pegar" las máquinas de cada segmento de una expresión regular con el fin de formar una máquina que corresponde a la expresión completa. De este modo, la construcción es inductiva, y sigue la estructura de la definición de una expresión regular: mostramos un NFA para cada expresión regular básica y posteriormente mostramos cómo se puede conseguir cada operación de expresión regular al conectar entre sí los NFA de las subexpresiones (suponiendo que éstas ya se han construido).

Expresiones regulares básicas Una expresión regular básica es de la forma **a**, $\boldsymbol{\varepsilon}$ o $\boldsymbol{\phi}$, donde **a** representa una correspondencia con un carácter simple del alfabeto, $\boldsymbol{\varepsilon}$ representa una coincidencia con la cadena vacía y $\boldsymbol{\phi}$ representa la correspondencia con ninguna cadena. Un NFA que es equivalente a la expresión regular **a** (es decir, que acepta precisamente aquellas cadenas en su lenguaje) es



De manera similar, un NFA que es equivalente a ε es

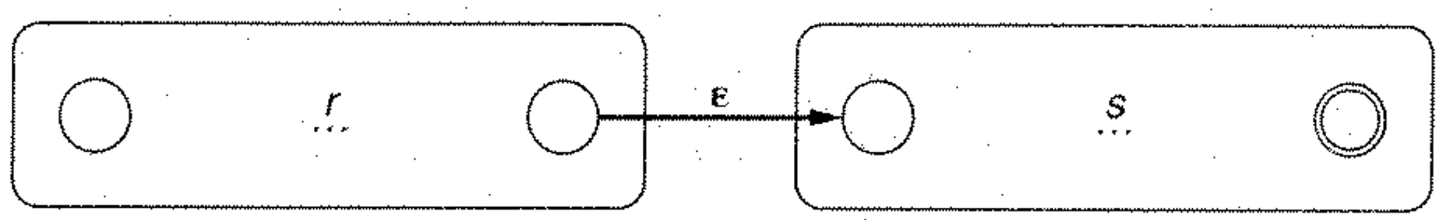


El caso de la expresión regular ϕ (que nunca ocurre en la práctica en un compilador) se deja como ejercicio.

Concatenación Deseamos construir un NFA equivalente a la expresión regular *rs*, donde *r* y *s* son expresiones regulares. Suponemos (de manera inductiva) que los NFA equivalentes a *r* y *s* ya se construyeron. Expresamos esto al escribir

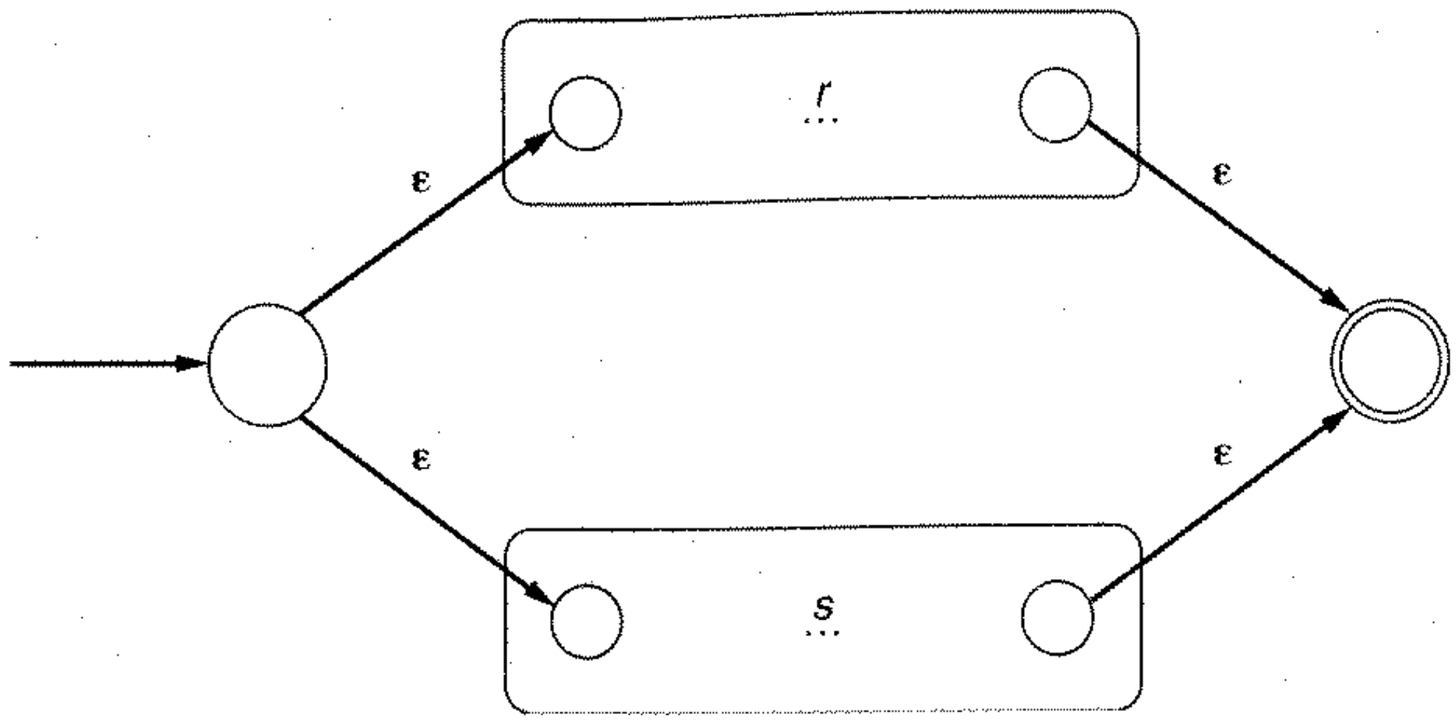
para el NFA correspondiente a *r*, y de manera similar para *s*. En este dibujo, el círculo a la izquierda dentro del rectángulo redondeado indica el estado de inicio, el círculo doble a la derecha indica el estado de aceptación y los tres puntos indican los estados y transiciones dentro del NFA que no se muestran. Esta figura supone que el NFA correspondiente a *r* tiene sólo un estado de aceptación. Esta suposición se justificará si todo NFA que construyamos tiene un estado de aceptación. Esto es verdadero para los NFA de expresiones regulares básicas, y será cierto para cada una de las construcciones siguientes.

Ahora podemos construir un NFA correspondiente a *rs* de la manera siguiente:



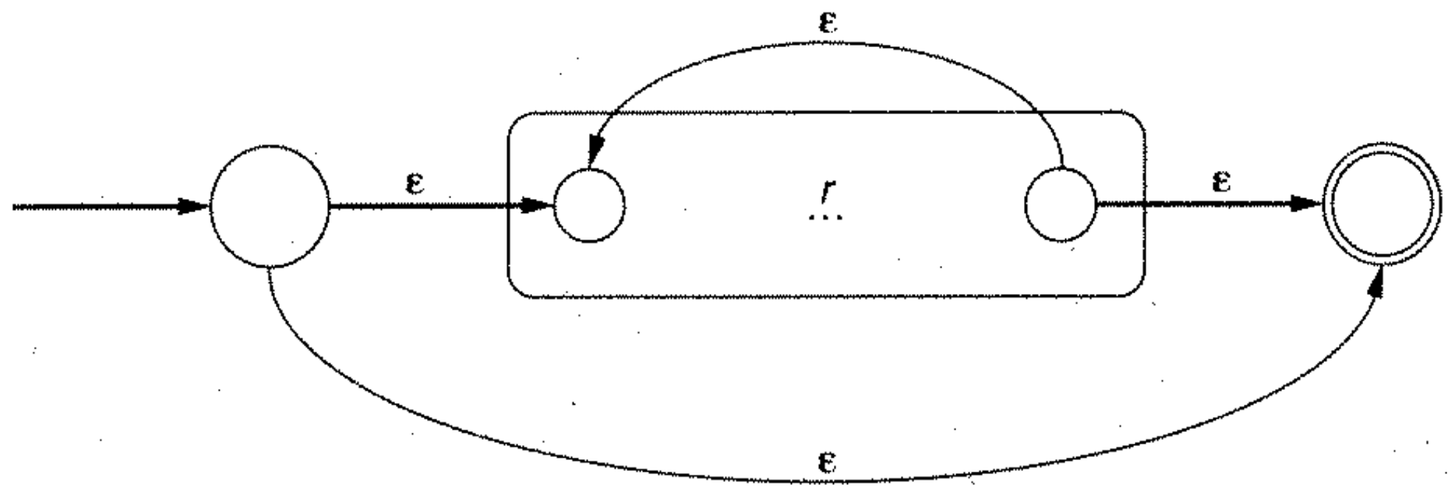
Conectamos el estado de aceptación de la máquina de *r* al estado de inicio de la máquina de *s* mediante una transición ε. La nueva máquina tiene el estado de inicio de la máquina de *r* como su estado de inicio y el estado de aceptación de la máquina de *s* como su estado de aceptación. Evidentemente, esta máquina acepta $L(rs) = L(r)L(s)$ y de este modo corresponde a la expresión regular *rs*.

Selección entre alternativas Deseamos construir un NFA correspondiente a $r \mid s$ bajo las mismas suposiciones que antes. Hacemos esto como se ve a continuación:



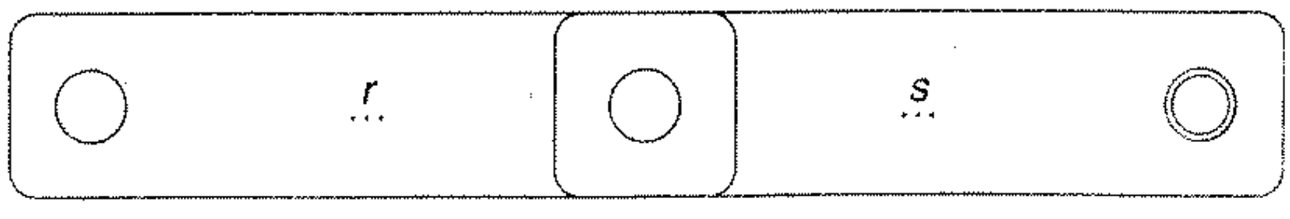
Agregamos un nuevo estado de inicio y un nuevo estado de aceptación y los conectamos como se muestra utilizando transiciones ε. Evidentemente, esta máquina acepta el lenguaje $L(r|s) = L(r) \cup L(s)$.

Repetición Queremos construir una máquina que corresponda a r^*, dada una máquina que corresponda a r. Haremos esto como se muestra enseguida:



Aquí nuevamente agregamos dos nuevos estados, un estado de inicio y un estado de aceptación. La repetición en esta máquina la proporciona la nueva transición ε del estado de aceptación de la máquina de r a su estado de inicio. Esto permite a la máquina de r ser atravesada una o más veces. Para asegurar que la cadena vacía también es aceptada (correspondiente a las cero repeticiones de r), también debemos trazar una transición ε desde el nuevo estado de inicio hasta el nuevo estado de aceptación.

Esto completa la descripción de la construcción de Thompson. Advertimos que esta construcción no es única. En particular, otras construcciones son posibles cuando se traducen operaciones de expresión regular en NFA. Por ejemplo, al expresar la concatenación rs, podríamos haber eliminado la transición ε entre las máquinas de r y s e identificar en su lugar el estado de aceptación de la máquina de r con el estado de inicio de la máquina de s, como se ilustra a continuación:



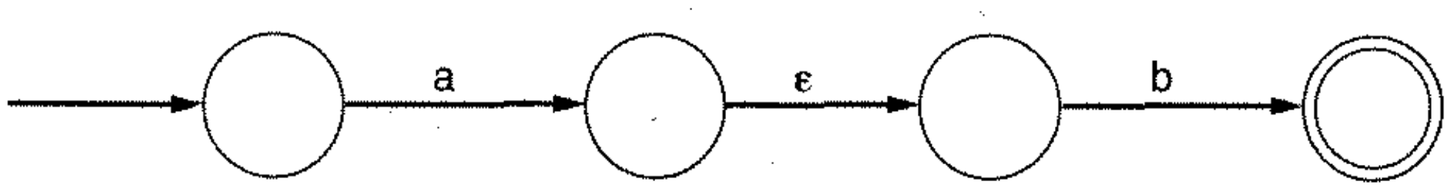
(Sin embargo, esta simplificación depende del hecho que en las otras construcciones el estado de aceptación no tiene transiciones desde ella misma hacia otros estados: véanse los ejercicios.) Otras simplificaciones son posibles en los otros casos. La razón por la que expresamos las traducciones a medida que las tenemos es que las máquinas se construyen según reglas muy simples. En primer lugar, cada estado tiene como máximo dos transiciones, y si hay dos, ambas deben ser transiciones ε. En segundo lugar, ningún estado se elimina una vez que se construye, y ninguna transición se modifica, excepto para la adición de transiciones desde el estado de aceptación. Estas propiedades hacen muy fácil automatizar el proceso.

Concluimos el análisis de la construcción de Thompson con algunos ejemplos.

Ejemplo 2.12

Traduciremos la expresión regular `ab|a` en un NFA de acuerdo con la construcción de Thompson. Primero formamos las máquinas para las expresiones regulares básicas `a` y `b`:

Entonces formamos la máquina para la concatenación `ab`:



Ahora formamos otra copia de la máquina para `a` y utilizamos la construcción para selección con el fin de obtener el NFA completo para `ab|a` que se muestra en la figura 2.8.

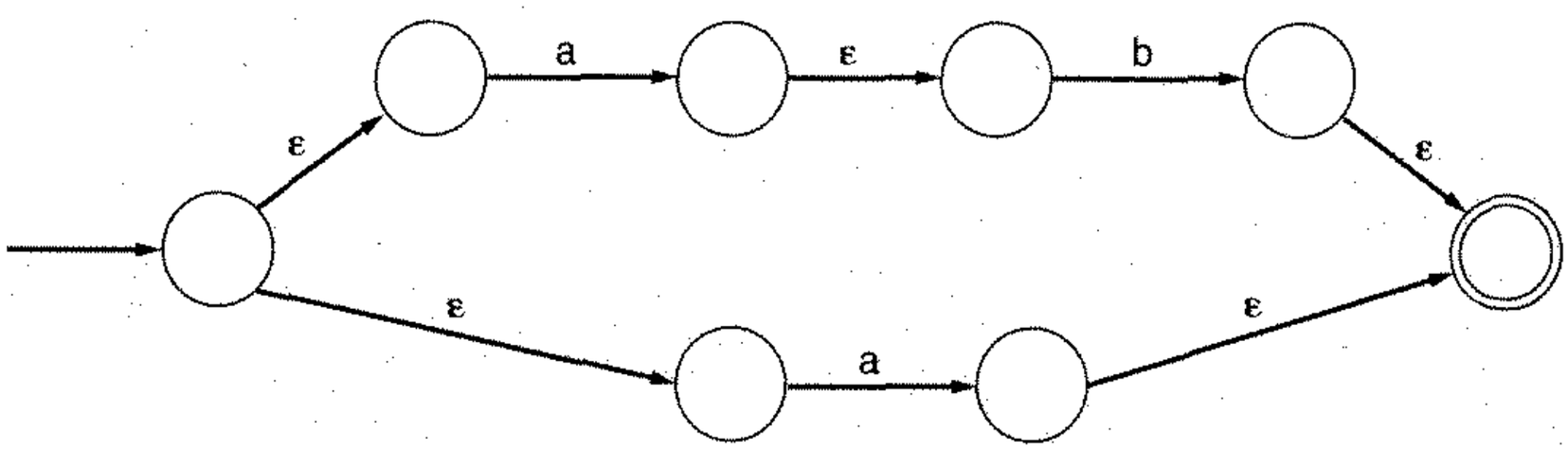
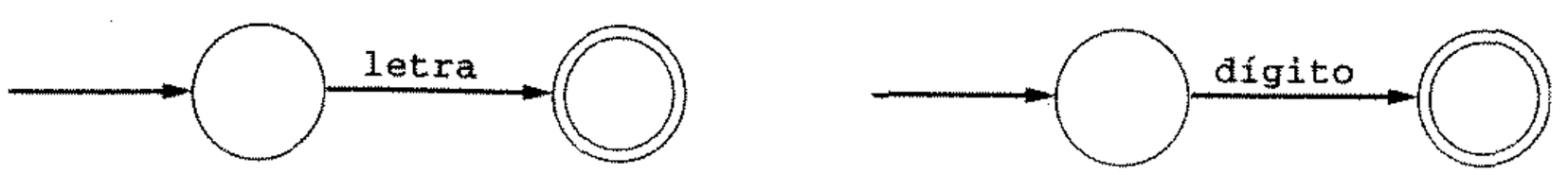


Figura 2.8
NFA para la expresión regular `ab|a` utilizando la construcción de Thompson

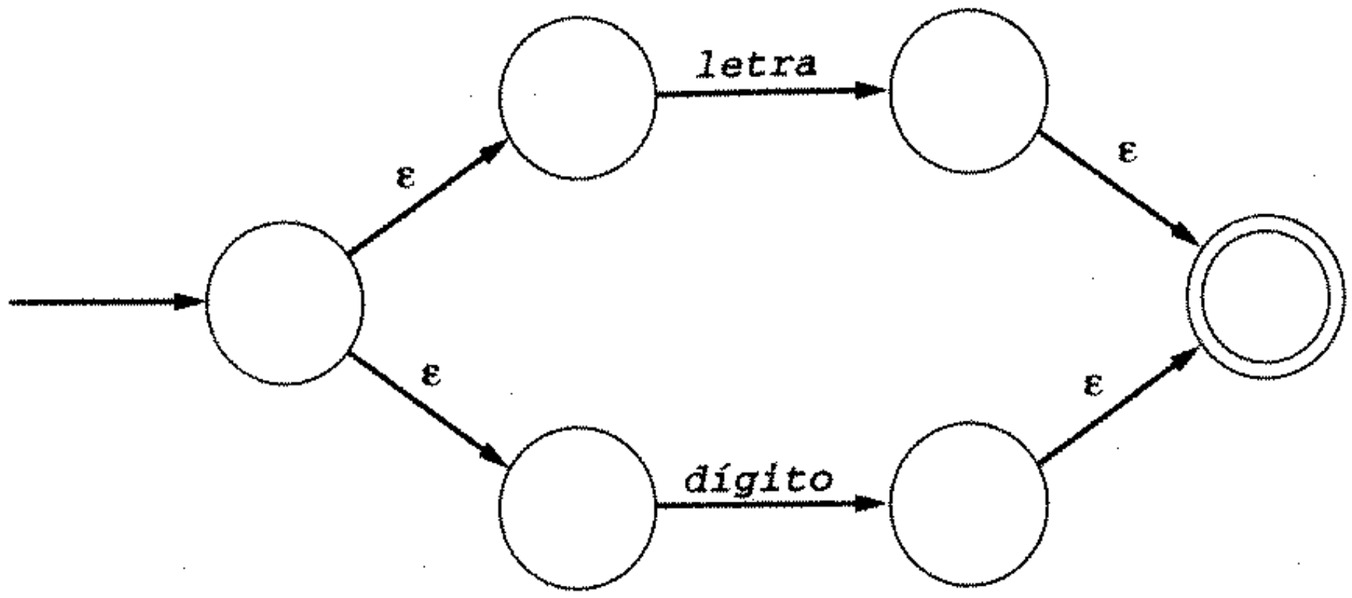
§

Ejemplo 2.13

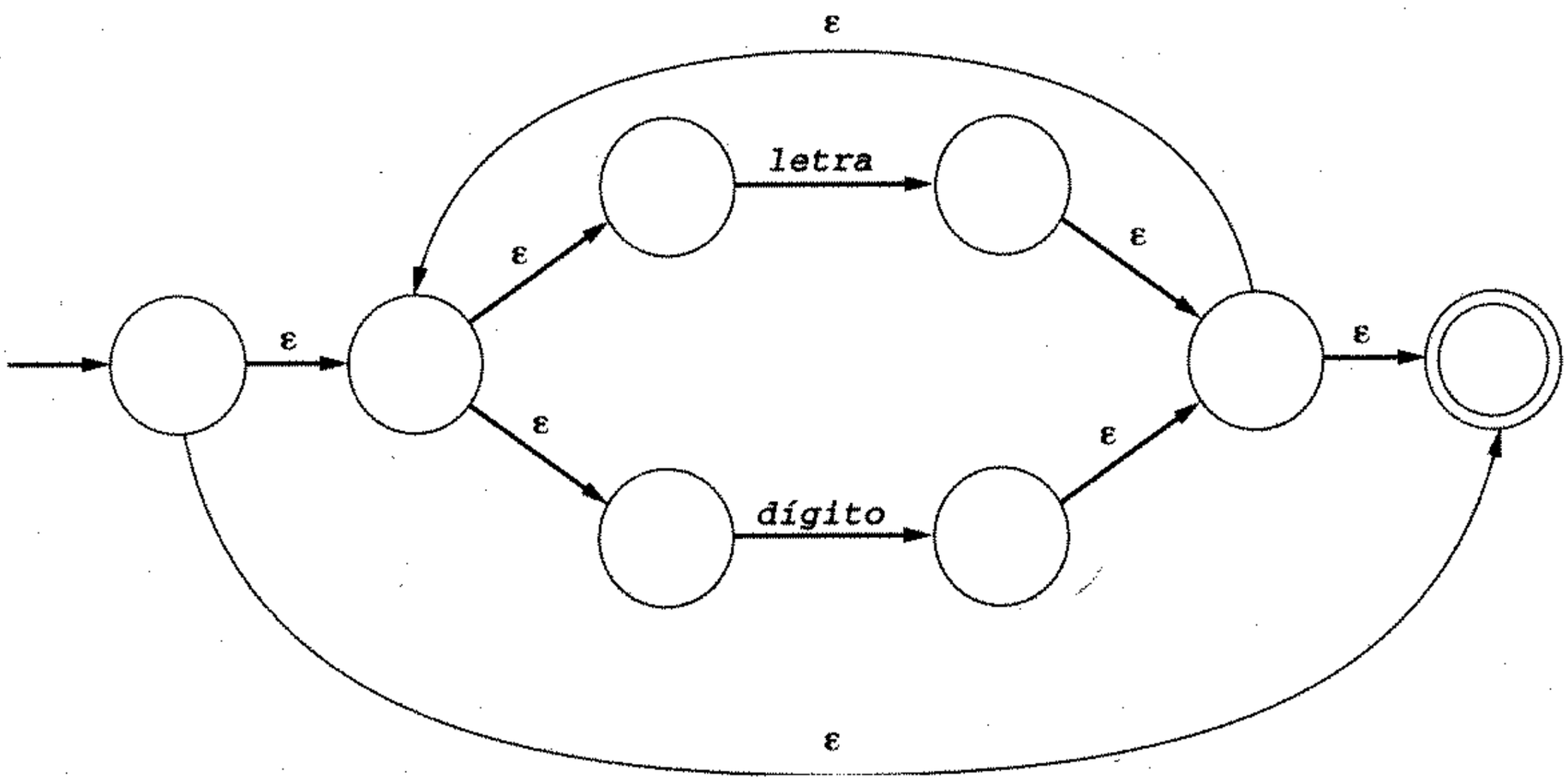
Formamos el NFA de la construcción de Thompson para la expresión regular `letra(letra|dígito)*`. Como en el ejemplo anterior, formamos las máquinas para las expresiones regulares `letra` y `dígito`:



Después formamos la máquina para la selección ***letra*|*dígito***:



Ahora formamos la NFA para la repetición **(*letra*|*dígito*)*** como se ve a continuación:



Por último construimos la máquina para la concatenación de ***letra*** con **(*letra*|*dígito*)*** para obtener el NFA completo, como se ilustra en la figura 2.9.

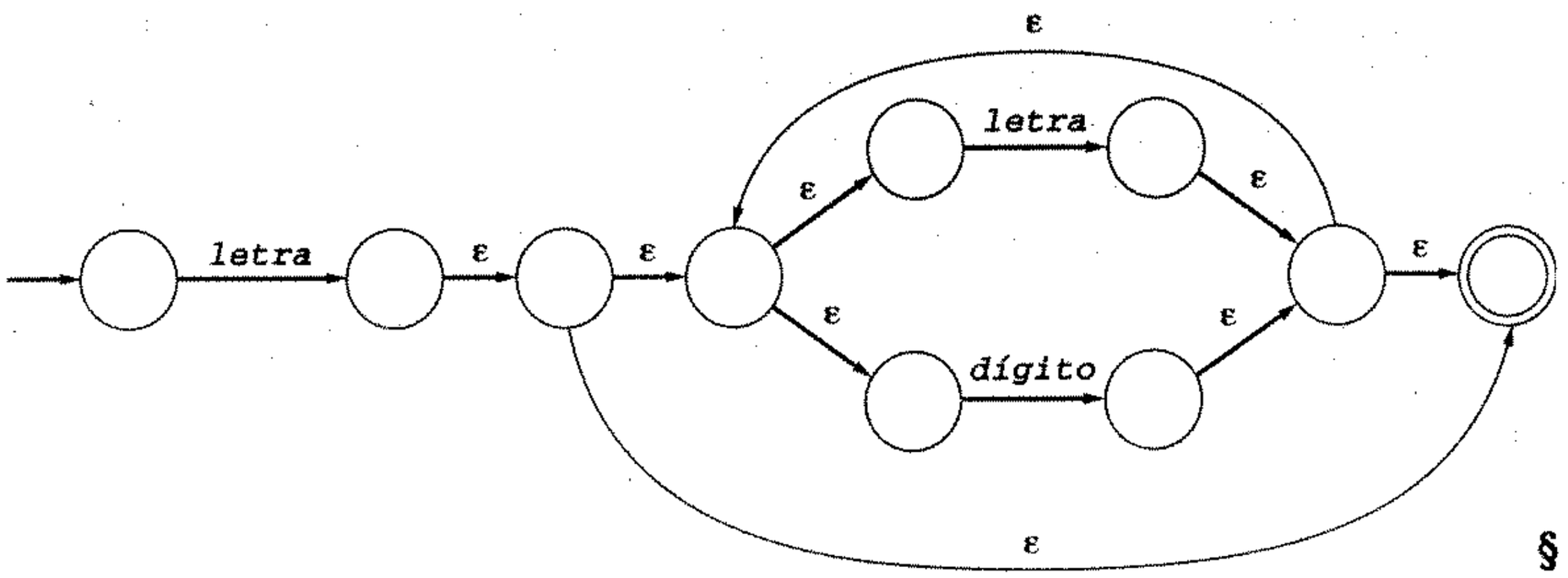


Figura 2.9
NFA para la expresión regular `letra-(letra|dígito)*` utilizando la construcción de Thompson

§

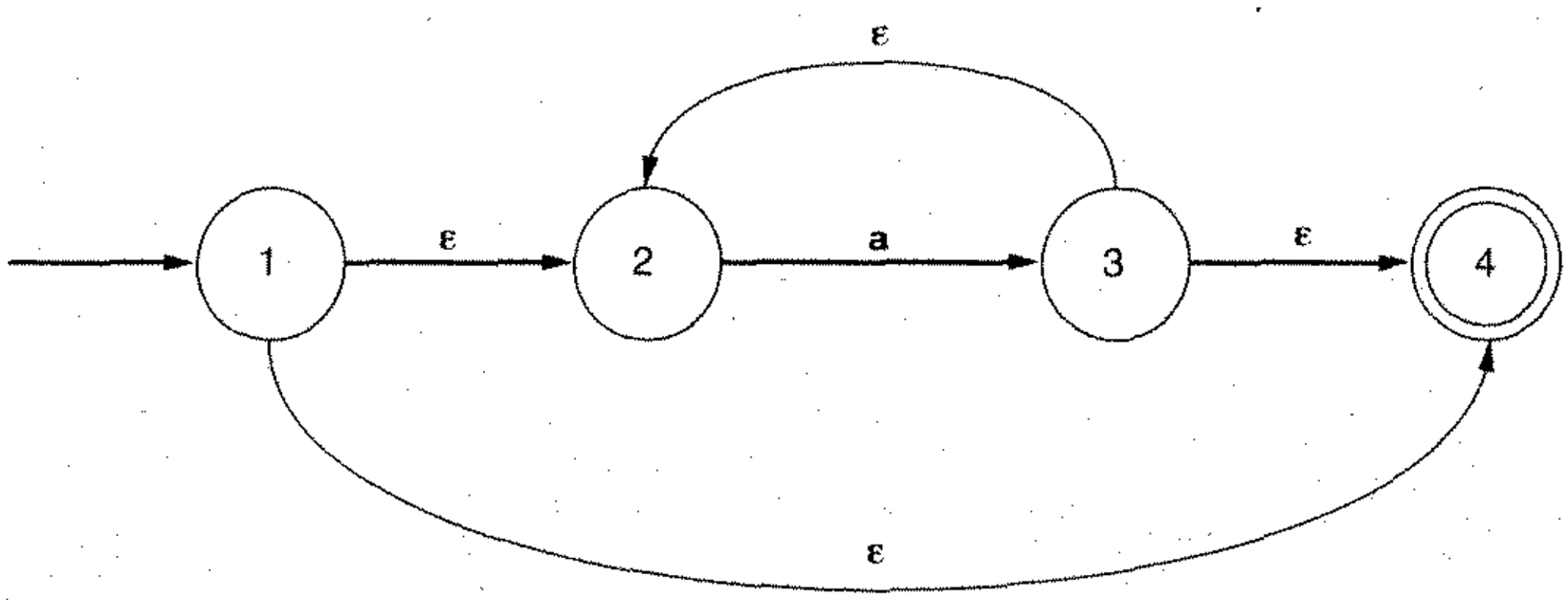
Como ejemplo final, observamos que el NFA del ejemplo 2.11 (sección 2.3.2) corresponde exactamente a la expresión regular `(a|c)*b` bajo la construcción de Thompson.

2.4.2 Desde un NFA hasta un DFA

Ahora deseamos describir un algoritmo que, dado un NFA arbitrario, construirá un DFA equivalente (es decir, uno que acepte precisamente las mismas cadenas). Para hacerlo necesitaremos algún método con el que se eliminen tanto las transiciones ε como las transiciones múltiples de un estado en un carácter de entrada simple. La eliminación de las transiciones ε implica el construir **cerraduras ε**, las cuales son el conjunto de todos los estados que pueden alcanzar las transiciones ε desde un estado o estados. La eliminación de transiciones múltiples en un carácter de entrada simple implica mantenerse al tanto del conjunto de estados que son alcanzables al igualar un carácter simple. Ambos procesos nos conducen a considerar conjuntos de estados en lugar de estados simples. De este modo, no es ninguna sorpresa que el DFA que construimos tenga como sus estados los *conjuntos de estados* del NFA original. En consecuencia, este algoritmo se denomina **construcción de subconjuntos**. Primero analizaremos la cerradura ε con un poco más de detalle y posteriormente continuaremos con una descripción de la construcción de subconjuntos.

La cerradura ε de un conjunto de estados Definimos la cerradura ε de un estado simple s como el conjunto de estados alcanzables por una serie de cero o más transiciones ε, y escribimos este conjunto como $\bar{s}$. Dejaremos una afirmación más matemática de esta definición para un ejercicio y continuaremos directamente con un ejemplo. Sin embargo, advierta que la cerradura ε de un estado siempre contiene al estado mismo.

Ejemplo 2.14 Considere el NFA siguiente que corresponde a la expresión regular `a*` bajo la construcción de Thompson:



En este NFA tenemos $\bar{1} = \{1, 2, 4\}$, $\bar{2} = \{2\}$, $\bar{3} = \{2, 3, 4\}$ y $\bar{4} = \{4\}$. §

Ahora definiremos la cerradura ε de un conjunto de estados como la unión de las cerraduras ε de cada estado individual. En símbolos, si S es un conjunto de estados, entonces tenemos que

$$\bar{S} = \bigcup_{s \text{ en } S} \bar{s}$$

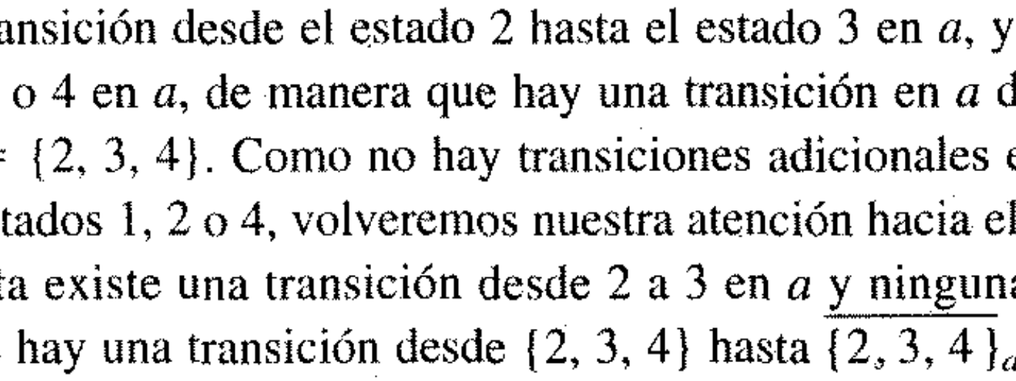
Por ejemplo, en el NFA del ejemplo 2.14, $\overline{\{1, 3\}} = \overline{1} \cup \overline{3} = \{1, 2, 4\} \cup \{2, 3, 4\} = \{1, 2, 3, 4\}$.

La construcción del subconjunto Ahora estamos en posición de describir el algoritmo para la construcción de un DFA a partir de un NFA M dado, al que denominaremos $\overline{M}$. Primero calculamos la cerradura ε del estado de inicio de M; esto se convierte en el estado de inicio de $\overline{M}$. Para este conjunto, y para cada conjunto subsiguiente, calculamos las transiciones en los caracteres a como sigue. Dado un conjunto S de estados y un carácter a en el alfabeto, calculamos el conjunto $S'_a = \{\ t \mid$ para alguna s en S existe una transición de s a t en $a\ \}$. Después calculamos $\overline{S'_a}$, la cerradura ε de S'_a. Esto define un nuevo estado en la construcción del subconjunto, junto con una nueva transición $S \xrightarrow{a} \overline{S'_a}$. Continuamos con este proceso hasta que no se crean nuevos estados o transiciones. Marcamos como de aceptación aquellos estados construidos de esta manera que contengan un estado de aceptación de M. Éste es el DFA $\overline{M}$. No contiene transiciones ε debido a que todo estado es construido como una cerradura ε. Contiene como máximo una transición desde un estado en un carácter a porque cada nuevo estado se construye de *todos* los estados de M alcanzables por transiciones desde un estado en un carácter simple a.

Ilustraremos la construcción del subconjunto con varios ejemplos.

Ejemplo 2.15

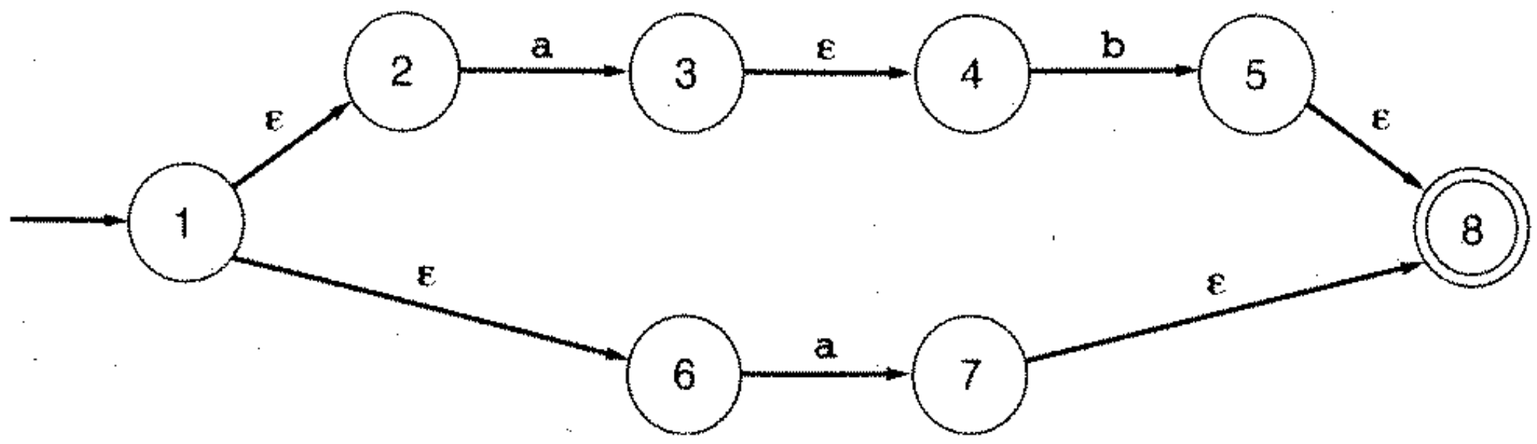
Considere el NFA del ejemplo 2.14. El estado de inicio del DFA correspondiente es $\overline{1} = \{1, 2, 4\}$. Existe una transición desde el estado 2 hasta el estado 3 en a, y no hay transiciones desde los estados 1 o 4 en a, de manera que hay una transición en a desde $\{1, 2, 4\}$ hasta $\overline{\{1, 2, 4\}_a} = \overline{\{3\}} = \{2, 3, 4\}$. Como no hay transiciones adicionales en un carácter desde cualquiera de los estados 1, 2 o 4, volveremos nuestra atención hacia el nuevo estado $\{2, 3, 4\}$. De nueva cuenta existe una transición desde 2 a 3 en a y ninguna transición a desde 3 o 4, de modo que hay una transición desde $\{2, 3, 4\}$ hasta $\overline{\{2, 3, 4\}_a} = \overline{\{3\}} = \{2, 3, 4\}$. De manera que existe una transición a desde $\{2, 3, 4\}$ hacia sí misma. Agotamos los estados a considerar, y así construimos el DFA completo. Sólo resta advertir que el estado 4 del NFA es de aceptación, y puesto que tanto $\{1, 2, 4\}$ como $\{2, 3, 4\}$ contienen al 4, ambos son estados de aceptación del DFA correspondiente. Dibujamos el DFA que construimos como sigue, donde nombramos a los estados mediante sus subconjuntos:



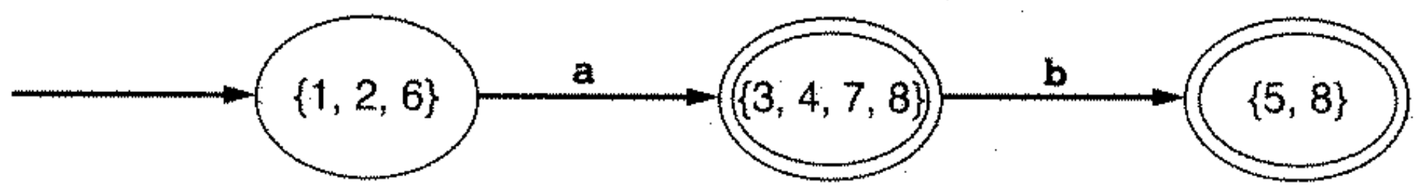
(Una vez que se termina la construcción, podríamos descartar la terminología de subconjuntos si así lo deseamos.) §

Ejemplo 2.16

Consideremos el NFA de la figura 2.8, al cual agregaremos números de estado:

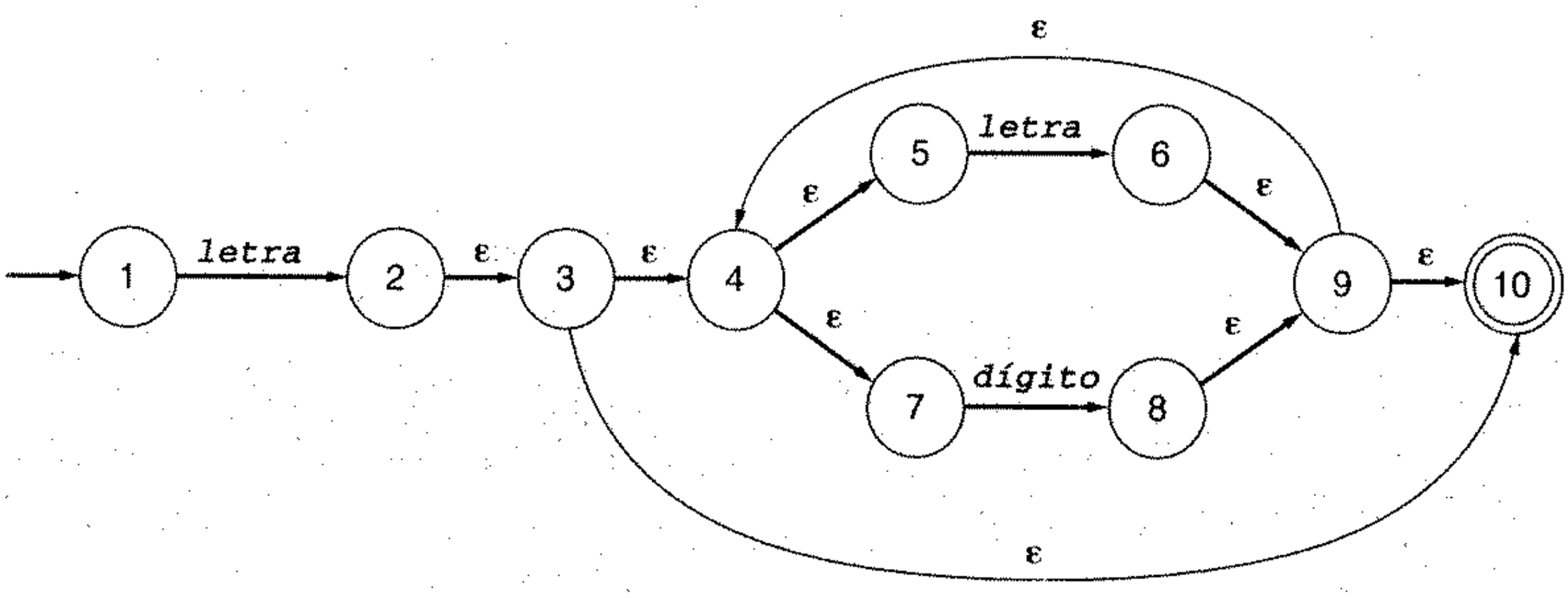


La construcción del subconjunto de DFA tiene como su estado de inicio $\overline{\{1\}} = \{1, 2, 6\}$. Existe una transición en *a* desde el estado 2 hasta el estado 3, y también desde el estado 6 hasta el estado 7. De este modo, $\{1, 2, 6\}_a = \overline{\{3, 7\}} = \{3, 4, 7, 8\}$, y tenemos $\{1, 2, 6\} \xrightarrow{a} \{3, 4, 7, 8\}$. Como no hay otras transiciones de carácter desde 1, 2 o 6, vayamos a {3, 4, 7, 8}. Existe una transición en *b* desde 4 hasta 5 y $\{3, 4, 7, 8\}_b = \overline{\{5\}} = \{5, 8\}$, y tenemos la transición $\{3, 4, 7, 8\} \xrightarrow{b} \{5, 8\}$. No hay otras transiciones. Así que la construcción del subconjunto produce el siguiente DFA equivalente al anterior NFA:

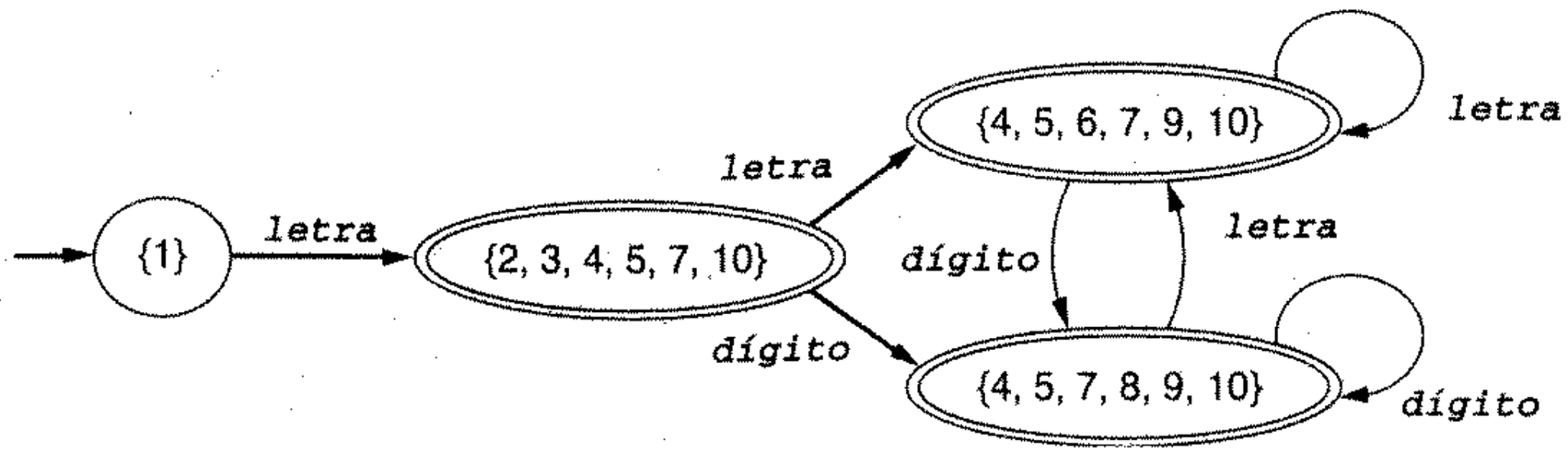


§

Ejemplo 2.17 Considere el NFA de la figura 2.9 (la construcción de Thompson para la expresión regular `letra(letra|dígito)*`):



La construcción del subconjunto continúa como se explica a continuación. El estado de inicio es $\overline{\{1\}} = \{1\}$. Existe una transición en ***letra*** para $\overline{\{2\}} = \{2, 3, 4, 5, 7, 10\}$. Desde este estado existe una transición en ***letra*** para $\overline{\{6\}} = \{4, 5, 6, 7, 9, 10\}$ y una transición en ***dígito*** para $\overline{\{8\}} = \{4, 5, 7, 8, 9, 10\}$. Finalmente, cada uno de estos estados también tiene transiciones en ***letra*** y ***dígito***, ya sea hacia sí mismos o hacia el otro. El DFA completo se ilustra en la siguiente figura:



2.4.3 Simulación de un NFA utilizando la construcción de subconjuntos

En la sección anterior analizamos brevemente la posibilidad de escribir un programa para simular un NFA, una cuestión que requiere tratar con la no determinación, o naturaleza no algorítmica, de la máquina. Una manera de simular un NFA es utilizar la construcción de subconjuntos, pero en lugar de construir todos los estados del DFA asociado, construimos solamente el estado en cada punto que indica el siguiente carácter de entrada. De este modo, construimos sólo aquellos conjuntos de estados que se presentan en realidad en una trayectoria a través del DFA que se toma en la cadena de entrada proporcionada. La ventaja de esto es que podemos no necesitar construir el DFA completo. La desventaja es que, si la trayectoria contiene bucles, un estado se puede construir muchas veces.

Pongamos por caso el ejemplo 2.16, donde si tenemos la cadena de entrada compuesta del carácter simple *a*, construiremos el estado de inicio {1, 2, 6} y luego el segundo estado {3, 4, 7, 8} hacia el cual nos movemos e igualamos la *a*. Entonces, como no hay *b* a continuación, aceptamos sin generar incluso el estado {5, 8}.

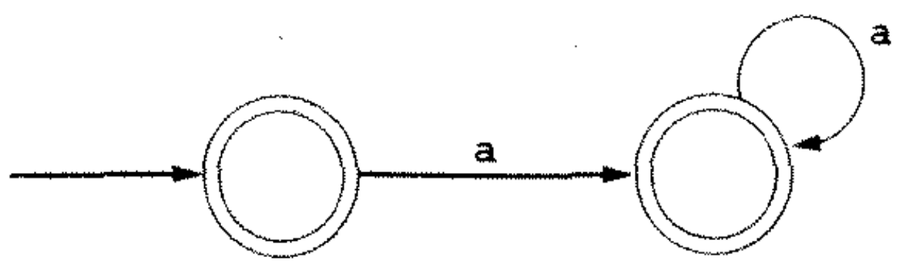
Por otra parte, en el ejemplo 2.17, dada la cadena de entrada *r2d2*, tenemos la siguiente secuencia de estados y transiciones:

$$\{1\} \xrightarrow{\texttt{r}} \{2, 3, 4, 5, 7, 10\} \xrightarrow{\texttt{2}} \{4, 5, 7, 8, 9, 10\}$$
$$\xrightarrow{\texttt{d}} \{4, 5, 6, 7, 9, 10\} \xrightarrow{\texttt{2}} \{4, 5, 7, 8, 9, 10\}$$

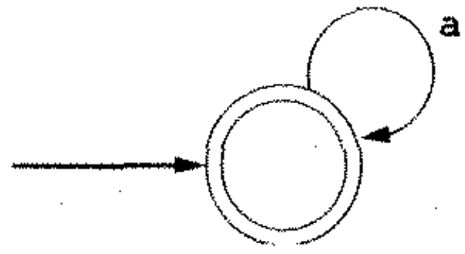
Si estos estados se construyen a medida que se presentan las transiciones, entonces todos los estados del DFA están construidos y el estado {4, 5, 7, 8, 9, 10} incluso se construyó dos veces. Así que este proceso es menos eficiente que construir el DFA completo en primer lugar. Es por esto que no se realiza la simulación de los NFA en analizadores léxicos. Queda una opción para igualar patrones en programas de búsqueda y editores, donde el usuario puede dar las expresiones regulares de manera dinámica.

2.4.4 Minimización del número de estados en un DFA

El proceso que describimos para derivar un DFA de manera algorítmica a partir de una expresión regular tiene la desafortunada propiedad de que el DFA resultante puede ser más complejo de lo necesario. Como un caso ilustrativo, en el ejemplo 2.15 derivamos el DFA



para la expresión regular `a*`, mientras que el DFA



también lo hará. Como la eficiencia es muy importante en un analizador léxico, nos gustaría poder construir, si es posible, un DFA que sea mínimo en algún sentido. De hecho, un resultado importante de la teoría de los autómatas establece que, dado cualquier DFA, existe un DFA equivalente que contiene un número mínimo de estados, y que este DFA de estados mínimos o mínimo es único (excepto en el caso de renombrar estados). También es posible obtener directamente este DFA mínimo de cualquier DFA dado, y aquí describiremos brevemente el algoritmo, sin demostrar que en realidad construye el DFA mínimo equivalente (debería ser fácil para el lector convencerse de esto de manera informal al leer el algoritmo).

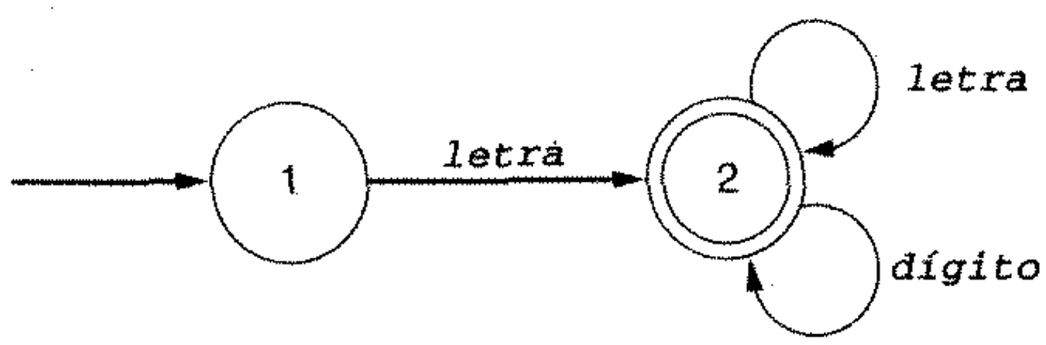
El algoritmo funciona al crear conjuntos de estados que se unificarán en estados simples. Comienza con la suposición más optimista posible: crea dos conjuntos, uno compuesto de todos los estados de aceptación y el otro compuesto de todos los estados de no aceptación. Dada esta partición de los estados del DFA original, considere las transiciones en cada carácter *a* del alfabeto. Si todos los estados de aceptación tienen transiciones en *a* para estados de aceptación, entonces esto define una transición *a* desde el nuevo estado de aceptación (el conjunto de todos los estados de aceptación antiguos) hacia sí misma. De manera similar, si todos los estados de aceptación tienen transiciones en *a* hacia estados de no aceptación, entonces esto define una transición *a* desde el nuevo estado de aceptación hacia el nuevo estado de no aceptación (el conjunto de todos los estados de no aceptación antiguos). Por otra parte, si hay dos estados de aceptación *s* y *t* que tengan transiciones en *a* que caigan en diferentes conjuntos, entonces ninguna transición *a* se puede definir para este agrupamiento de los estados. Decimos que *a* **distingue** los estados *s* y *t*. En este caso, el conjunto de estados que se está considerando (es decir, el conjunto de todos los estados de aceptación) se debe dividir de acuerdo con el lugar en que caen sus transiciones *a*. Por supuesto, declaraciones similares se mantienen para cada uno de los otros conjuntos de estados, y una vez que hemos considerado todos los caracteres del alfabeto, debemos movernos a ellos. Por supuesto, si cualquier conjunto adicional se divide, debemos regresar y repetir el proceso desde el principio. Continuamos este proceso de refinar la partición de los estados del DFA original en conjuntos hasta que todos los conjuntos contengan sólo un elemento (en cuyo caso, mostramos que el DFA original es mínimo) o hasta que no se presente ninguna división adicional de conjuntos.

Para que el proceso que acabamos de describir funcione correctamente, también debemos considerar transiciones de error a un estado de error que es de no aceptación. Esto es, si existen estados de aceptación *s* y *t*, tales que *s* tenga una transición *a* hacia otro estado de aceptación, mientras que *t* no tenga ninguna transición *a* (es decir, una transición de error), entonces *a* distingue *s* y *t*. De la misma manera, si un estado de no aceptación *s* tiene una transición *a* hacia un estado de aceptación, mientras que otro estado de no aceptación *t* no tiene transición *a*, entonces *a* distingue *s* y *t* también en este caso.

Concluiremos nuestro análisis de la minimización de estado con un par de ejemplos.

Ejemplo 2.18

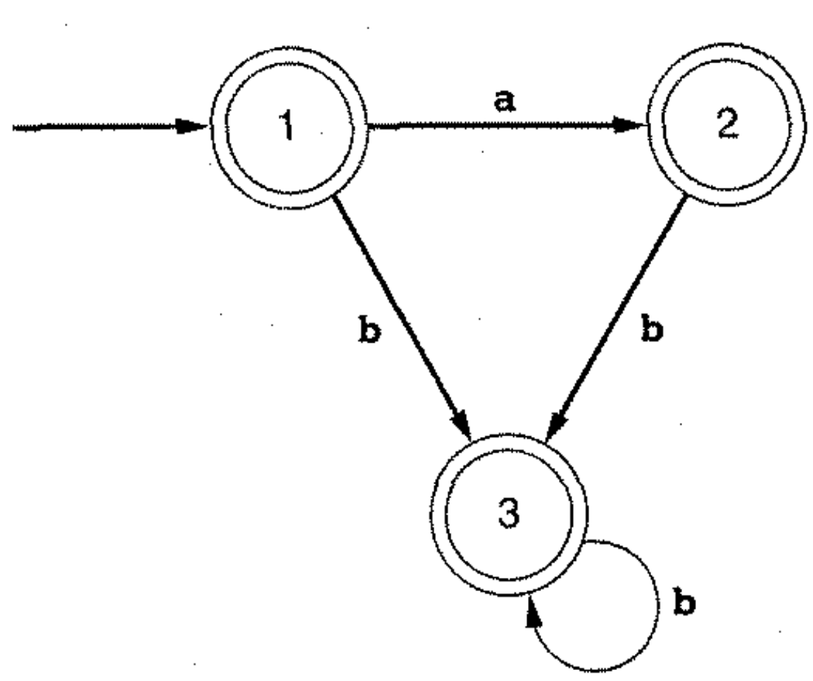
Consideremos el DFA que construimos en el ejemplo anterior, correspondiente a la expresión regular *letra*(*letra*|*dígito*)*. Tenía cuatro estados compuestos del estado inicial y tres estados de aceptación. Los tres estados de aceptación tienen transiciones a otros estados de aceptación tanto en *letra* como en *dígito* y ninguna otra transición (sin errores). Por consiguiente, los tres estados de aceptación no se pueden distinguir por ningún carácter, y el algoritmo de minimización da como resultado la combinación de los tres estados de aceptación en uno, dejando el siguiente DFA mínimo (el cual ya vimos al principio de la sección 2.3):



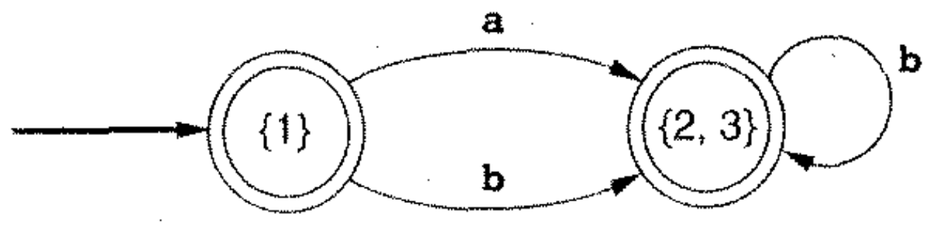
§

Ejemplo 2.19

Considere el siguiente DFA, que dimos en el ejemplo 2.1 (sección 2.3.2) como equivalente a la expresión regular (a|ε)b*:



En este caso todos los estados (excepto el estado de error) son de aceptación. Consideremos ahora el carácter *b*. Cada estado de aceptación tiene una transición *b* a otro estado de aceptación, de modo que ninguno de los estados se distingue por *b*. Por otro lado, el estado 1 tiene una transición *a* hacia un estado de aceptación, mientras que los estados 2 y 3 no tienen transición *a* (o, mejor dicho, una transición de error en *a* hacia el estado de no aceptación de error). Así, *a* distingue el estado 1 de los estados 2 y 3, y debemos repartir los estados en los conjuntos {1} y {2, 3}. Ahora comenzamos de nuevo. El conjunto {1} no se puede dividir más, así que ya no lo consideraremos. Ahora los estados 2 y 3 no se pueden distinguir por *a* o *b*. De esta manera, obtenemos el DFA de estado mínimo:



§